Aalto University
School of Engineering

Tuomas Lempiäinen

Test automation on a third party mobile software

Thesis of diploma given in for the degree
of Master of Science in Technology

Espoo 26.5.2015
Instructor: Tuomo Hakulinen, RAY
Supervisor: Professor Kalevi Ekman

Aalto University
School of Engineering

Aalto University, P.O. BOX 11000, 00076
AALTO
www.aalto.fi

Abstract of master's thesis

Author Tuomas Lempiäinen	
Title of thesis Test automation on a third party mobile software	
Department Department of Engineering Design and Production	
Professorship Engineering Design	**Code of professorship** K3001
Thesis supervisor Professor Kalevi Ekman	
Thesis advisor Tuomo Hakulinen, Head of Testing	
Date 26.05.2015 **Number of pages** 55+19	**Language** English

Abstract

This work aimed to create viable test automation for Finland's Slot Machine Association's mobile casino software. The work contains a test plan for the software, and an introduction of the selected automation framework.

The software is developed by a third party. The results of this thesis indicate that testing third party software differs tremendously from testing in-house developed software. Instead of a multiphased testing process, only a one shot acceptance testing run can be performed on new builds. The third party development also had a major effect on the selection of the test automation framework. In this case, SeeTest Automation by Experitest combined with Robot Framework ended up being the best solution for automation framework.

Although the mobile casino runs on iOS, Android and Windows Phone, Windows Phone was decided to leave out from the test automation scope. The decision was based on the uncertainty of the Windows Phone support and the complexity of its automation.

Keywords

Test automation, software development, mobile software, third party development, testing, acceptance testing.

Aalto-yliopisto
Insinööritieteiden
korkeakoulu

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Tuomas Lempiäinen

Työn nimi Kolmannen osapuolen mobiiliohjelmiston testiautomaatio

Koulutusohjelma Konetekniikka

Pääaine Koneensuunnittelu **Koodi** K3001

Työn valvoja Professori Kalveli Ekman

Työn ohjaaja Testauksen Palvelupäällikkö Tuomo Hakulinen

Päivämäärä 26.05.2015 **Sivumäärä** 55+19 **Kieli** Englanti

Tiivistelmä

Tämän työn tarkoituksena oli luoda testiautomaatio ympäristö Raha-automaattiyhdistyksen mobiilikasinon laadunvarmistukseen. Työ käsittää tuotetta varten luodun testaussuunnitelman sekä valitun automaatioympäristön esittelyn. Testauksen kohteena oleva ohjelmisto on kolmannen osapuolen kehittämä. Työssä kävi ilmi, että kolmannen osapuolen ja itse kehitetyn ohjelmiston testaus eroavat toisistaan merkittävästi. Monivaiheisen testausprosessin sijaan kolmannen osapuolen ohjelmistolle voidaan suorittaa ainoastaan yksi hyväksymistestauskierros. Kolmannen osapuolen mukana ololla oli myös suuri merkitys automaatio-ohjelmiston valintaan. Työssä päädyttiin käyttämään Experitestin SeeTest Automation sekä Robot Framework -ohjelmistoja.

Vaikka mainittu mobiilikasino toimii sekä iOS, Android että Windows Phone -käyttöjärjestelmillä, päätettiin työssä jättää Windows Phone automaation ulkopuolelle. Syy ratkaisuun oli epävarmuus käyttöjärjestelmän tulevaisuudesta sekä sen automaation mukana tuomat haasteet.

Avainsanat

Testaus, testiautomaatio, mobiiliohjelmisto, kolmannen osapuolen kehitys, hyväksymistestaus

Preface

This work was written for the benefit of the testing department of Finland's Slot Machine Association (RAY). I want to thank my supervisor at RAY Tuomo Hakulinen, Head of Testing, for the opportunity to write this research. Also I would like to gratefully acknowledge my instructor Kalevi Ekman, Professor of Product Development at Aalto University, for his guidance and support.

From RAY, I want to thank Testing Specialist Juuso Issakainen; Test Automation Specialists Juho Valonen, Saku Palanne and Jaakko Koho; Test Managers Satu Ilmonen and Ville Välimaa; Head of Digital Operations Petri Suominen and Mobile Game Client Specialist Jussi Gustafsson. Also special thanks to Salla Mölsä and Jari Pitkänen for proof-reading the text.

Espoo 26.5.2015

Tuomas Lempiäinen

Content

Abbreviations

API	Application programming interface
GUI	Graphical user interface
GUITAR	GUI Testing Framework
MBT	Model-based testing
MGP	Mobile gaming platform
NGM	New generation mobile
OCR	Optical character recognition
OS	Operating system
POC	Proof of concept
PT	Playtech
QA	Quality assurance
RAY	Finland's Slot Machine Association
RF	Robot Framework
RIDE	Robot Framework Development Environment
ROI	Return of investment
STA	SeeTest Automation
SUT	System under test
TDD	Test driven development
VNC	Virtual Network Computing
WP	Windows Phone

1 Introduction

1.1 Aim of the work

The main purpose of this work is to create a viable test automation set for Finland's Slot Machine Association's (RAY) mobile casino software. The automation set is to be used as a smoke and regression test run on new builds. So far, the contribution of RAY's quality assurance (QA) on mobile test automation has been minimal. The focus of developing test automation has concentrated on more business-critical systems, such as backing systems and slot machines. For the moment, all tests on the mobile platform are performed manually. Due to this, the work needs to be divided into three sections. Firstly, to ensure the powerful use of automation, the existing mobile testing strategy needs to be revised. Automation of existing manual test cases would most likely result in ineffective use of automation. In practice, the first step is to create a completely new testing strategy for the mobile platform. The strategy will contain not only the automation part but also the tests not to face automation.

The second part will consist of selecting proper automation tools. As no test automation has formerly been done on the mobile platform, no mobile compatible automation framework exists in the association. The decision of the framework is highly dependent on the test strategy created on the first step as different types of automation have different requirements for the framework. The third section consists of conducting a proof of concept (POC) with the chosen framework. The POC is realized to ensure that the chosen framework actually suits the problem at hand. If any doubts arise against the selected framework, the decision needs to be revised.

The primary methods applied to achieve the goal of the work are literature survey, expert consultations, benchmarking and experiments. As a rough guide line, the further the study proceeds the more focus is moved from literature inquiry towards more experimental methods.

1.2 RAY

Finland's Slot Machine Association has the sole right to casino games, slot machines and two casinos in Finland. The association was founded in 1938 to raise funds for Finnish social and healthcare organizations. This basic idea yet exists nowadays and all the profit made is distributed to these organizations and to the rehabilitation of war veterans. The games can be played in over 7300 sites – mostly located in partners' properties, such as stores, bars and gas stations, and RAY hosted arcades. In addition to these games that form the basis of RAY's operation, also online casino games are provided. (RAY 2014b)

RAY is a Finnish game house that employs 1631 (31.12.2013) people all around Finland in various tasks. Almost every part of its business is self-organized – from maintenance and R&D to customer service. Only a small part of the business has been outsourced. In 2013, the revenue of RAY was 791.4 million euros from which 413.3 million euros were distributed to the Finnish welfare. (RAY 2014b)

1.2.1 Digital channels

In the beginning of 2010, the Finnish Government allowed RAY to provide games also online. By the end of 2010, RAY launched casino games online, including table games,

slots and poker games. (RAY 2011) The games use a platform provided by Playtech Ltd (referred later also as PT). (RAY 2010). At present over 160 games can be played at www.ray.fi. The growth of online gaming has been remarkable. The volume has almost tripled since 2010. (RAY 2014b, RAY 2011) In 2013, the revenue of online games was 60 million euros which was 7.6 per cent of RAY's annual revenue. (RAY 2014b)

1.2.2 Mobile platform

In the beginning of 2013, RAY introduced games also for mobile devices. A screen capture of the mobile game's lobby can be seen in Figure 1. As in the desktop version, also the mobile games are provided by Playtech Ltd. Part of the games are RAY's own classics that Playtech has ported to their platform. The mobile casino launched with seven slot and table games. (RAY 2014b) Since then 13 new games have been released as the mobile casino provides now a total of 20 different games (RAY 2014a).

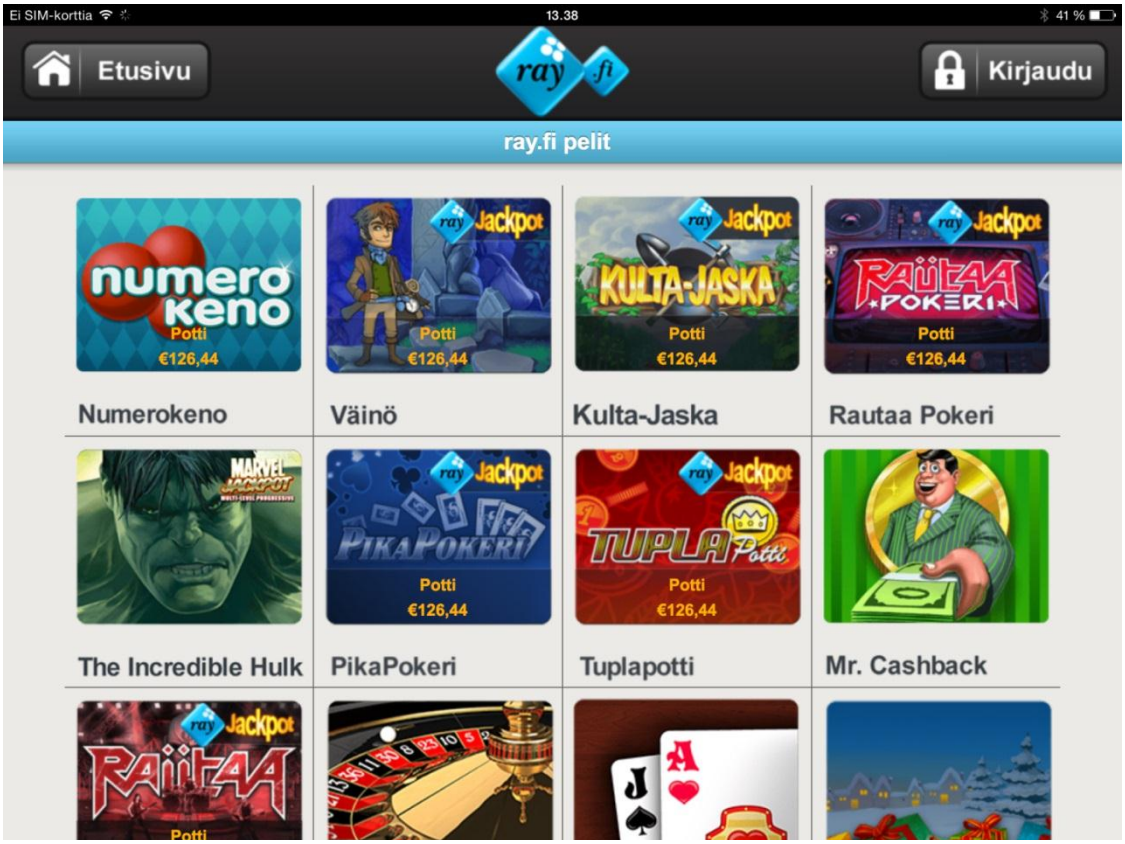

FIGURE 1 MOBILE GAME'S LOBBY VIEW ON AN IPAD

The mobile casino is a web based application that is supported on most devices including Windows Phone, iOS and Android operating systems (OS) (Playtech 2014). The casino can be divided into three different components: the website (later referred to as Lobby) and two different gaming platforms called New Generation Mobile (NGM) and Mobile Gaming Platform (NGM). Besides the games themselves, the website provides gaming related functionality such as deposits, withdraws, customer service, registration and responsible gaming tools.

The games are provided in two modes: fun mode and real money mode. In fun mode, the games are played with virtual money where player is not facing any monetary risk nor reward. The real mode in the other hand is played with real money. (RAY 2014a)

Although most of the games are available on the desktop version of the casino, they are completely different games beneath the surface. Visually and experience-wise the games seem alike, but otherwise the corresponding mobile and desktop games are completely different.

1.3 Motives for Quality Assurance

The main reason for quality assurance (QA) is simple: It is cheaper to prevent and search for product failures before production than to deal with customer complaints when a failure occurs in production. From a business point of view, raising the QA budget is justified if it can be shown that extra tests will save money in the long run. Quality-related costs are not only limited to the search and prevention of failures (prevention costs). They also include the internal costs (appraisal costs) and costs resulting from coping with the errors and failures of the product (internal and external failure costs). (Kaner, Falk et al. 1999)

The size of the online gambling industry is tremendous. It is predicted that in 2015 the size of the branch will reach 36 billion euros (Statista 2015). As the online casino gaming branch is highly competitive, the quality standards must be set high. Poor software quality results in loss of customers especially in web based services where alternatives exist (Whittaker 2009). In the gaming industry there, is a lot at stake for games that succeed because successful games can make huge revenue. (Schultz, Bryant et al. 2005) Since playing is all about making transactions with the money of customers, the reliability of the software must be uncompromised as it is stated in RAY's principles (RAY 2014b).

In online gaming, huge amounts of monetary transactions between accounts take place every day. Due to the great number of transactions, almost all of them are automated. In most cases, there is no one supervising these automated user-made transactions. Hence, the system needs to be flawless and leave no room for misuse. Faults in automated money transactions can easily lead to significant losses and decrease in reputation, as was seen in the Ålands Slot Machine Association's (PAF) case in 2006-2007 (Pokerisivut 2009). In one build of their online casino, the software enabled money transactions that were not deducted from the player's account but were visible in the targeted account. The misuse was not discovered until the player had withdrawn over 500.000 euros. (Pokerisivut 2009) On top of the lost money, such events generate mistrust towards the system among the users (O'Leary 2008). All of these consequences are types of external failure costs which are extremely harmful since they are visible to the customers (Kaner, Falk et al. 1999)

2 Methodology

This chapter introduces the theory behind the study. Chapters 2.1 and 2.2 focus on the theory and practices of testing software in general. Test automation, focusing on desktop software, will be discussed in Chapter 2.3. Chapter 2.4 takes a deep dive into mobile application testing. The theory of the test automation of mobile applications will also be discussed in this chapter.

Chapter 2.6 jumps to RAY's testing processes and the regulations under which RAY has to operate. The complex testing and development cycles of the mobile casino will also be under scrutiny.

2.1 Testing the software

2.1.1 Even software fail

Testing in general is a series of actions taken to generate knowledge on software quality. Another approach to testing is that it is performed to find problems in the software and getting them fixed (Kaner, Falk et al. 1999). Testing alone does not produce anything in itself. Testing is a support activity in the development process which is meaningless without development. (Hass 2008) By testing the product and fixing the occurred defects prior to release, the quality of the software is improved. If only few or no bugs are found during testing, confidence in the software quality is gained. Confidence and knowledge in the quality of the product are key factors in decision making. (ISTQB 2011)

The reasons behind software failures are various but human error stands behind most of them. Software is written by people, and since people make mistakes, the software is prone to defects. Also the tools utilized to create software are created by people and none of these are perfect. (Schultz, Bryant et al. 2005) As people's mistakes remain in the code, they are considered as defects. The defects cause no harm as long as they are not executed (i.e. the particular lines of the code are not executed) but when executed they will cause the software to fail. A failure is considered to be a deviation of the software's expected delivery or service. (Hass 2008).

Today, a software can consist of up to millions of lines of code which make the software design extremely complex (Schultz, Bryant et al. 2005). As the code becomes longer, also the complexity of the software also increases. Other reasons for complex systems are distributed and highly configurable systems. Moreover, systems divided into multiple subsystems can result in extended complexity. (Aho, Kanstén et al. 2014) Even if it were possible to create a perfect code, problems will occur at the latest when the software is used on a huge range of platforms and devices with multiple configurations. (Schultz, Bryant et al. 2005) The more complex the system, the more difficult it is to test.

2.1.2 Different phases of testing

Testing is usually done in multiple stages of the software development process. Testing phases themselves can be named in various ways, though the main principles are mostly the same. One way of dividing the phases is represented as follows (ISTQB 2011):

- **Unit tests** are usually performed by the developer for each function/class/module of the code to ensure the code's purity. One approach to

unit tests in test driven development is to script automated tests for the component prior to coding.

- **Integration tests** are performed to ensure the communication between two or more units. Units are different components of the software architecture. As the complexity of the system grows, the integration tests become more difficult since it is harder to locate the origins of the defects.
- **System testing** is the first step when the product is tested as a whole. Although in most cases system testing cannot be carried out in production environment, the test environment should be production-like. The objective of the test phase is to run the system against multiple setups and configurations, and to perform first use case tests. Typically, an independent test team is to carry out the phase.
- **Acceptance testing** is typically performed by the customer or users. At this phase, the main point is not finding defects in the system but rather establishing confidence in the system. The most critical defects ought to be found in the earlier phases. Acceptance testing can be split in to multiple sub levels including alpha and beta field tests.

Regression testing is a type of testing that some consider as a separate testing phase. It may refer to two different activities. One way of carrying out regression testing is to re-execute a test that found a defect on the last build of the software on a new build in order to verify whether a defect has been fixed or not. (Kaner, Falk et al. 1999) A more common definition is that a series of tests is performed on a new release of the software to ensure that the fixes in the update did not result in any undesired contamination (Kaner, Falk et al. 1999, Schultz, Bryant et al. 2005). Despite the multiple definitions, regression testing is a testing phase that is always an activity conducted on new releases of software.

Not all projects can use the described phases in the aforementioned order. Development projects are led in different manners that all have different needs, though the main principles always remain the same. One development ideology that could adapt the phases as dictated is the waterfall model. In the waterfall model, the product is developed ascending step by step downwards closer to a finished product. This model is though idealistic since all real world projects are at least up to some extent iterative. (Hyysalo 2009) These days agility is a common principle in software development. In an agile development team, the testers are also included in to the team to participate in the development process. Most agile teams focus on delivering small updates in rapid pace instead of seldom great updates. (Balasubramaniam 2014)

2.1.3 The “how” of testing

How to test the software? Of course, the chosen techniques depend on the test level and the project itself. There are multiple ways of dividing testing techniques and test types. One way of dividing test techniques is to separate them into structure- and specification-based testing also known as white-box and black-box testing respectively as shown in Figure 2.

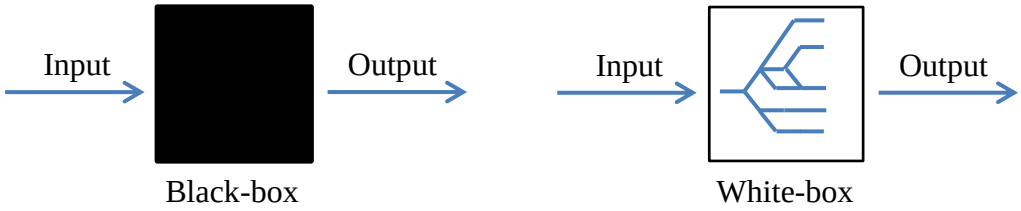

FIGURE 2. THE DIFFERENCE BETWEEN BLACK-BOX AND WHITE-BOX TESTING TECHNIQUES.

The key difference in these two techniques is in the knowledge of the system architecture. In white-box (also known as clear- or glassbox) techniques, testers use the knowledge of the system's source code when designing tests. Having the knowledge of the code, testers can focus the test cases on areas prone to defects. (Burnstein cop. 2003) Though white-box techniques are useful in all phases of testing, they are most useful in early stages (unit testing) of development. In early stages of development, the tests are narrowed down to small modules of the software. These tiny tests are worthwhile because executing and analyzing the tests is rapid. (ISTQB 2011, Burnstein cop. 2003) Code verification tests, like statement, code and decision coverage tests, are typical examples of early phase white-box techniques. They are used to ensure that the code covers all possible use cases and that all lines of the code are executed at some point. (ISTQB 2011)

Whereas white-box techniques enhance the knowledge of the code, black-box techniques rely on the specifications of the system. In all black-box techniques, testers only have the knowledge what the system does, not how it is done. The system is considered to be an opaque box. The only knowledge needed is what selected inputs should generate. (Burnstein cop. 2003) These input-output relations should be described as early in the process as possible, for example in the list of requirements. (Kaner, Falk et al. 1999)

A third testing technique combines the two models presented resulting in a technique known as grey-box testing. Basically grey-box techniques are like black-box techniques but they use some knowledge of the underlying code. The knowledge is only partial so they cannot be referred to as white-box techniques. Grey-box techniques may include reverse engineering on the software, for example on testing boundary values or error messages. (Khan 2010)

The art of exploratory testing or experience-based testing is a branch in software testing that incorporates the knowledge and experience of the tester (ISTQB 2011). Exploratory testing differs from scripted manual testing in that it does not include any particular steps of performing the tests. The way of execution or choice of steps relies on tester's experience. (Bach 2003) James Whittaker (2009) compares exploratory testing to the work of real explorers. Explorers wander in to new areas with preset goals. Their goal is to reach a new place but the exact path is defined on the way. Their mission is to reveal what lies beneath those untraveled paths. The better the explorer, the easier and faster the mission is accomplished. A better explorer makes better decisions on the way. The connection to software testing is that the tester is the explorer exploring the software. Like real explorers, a tester performing exploratory testing also has a preset goal to wander in a defined area of the software. Their mission is the same as in all testing: to gain understanding of the software and to find bugs. To maximize the gains, exploratory testing should be performed on functionalities that are complex or where bugs are likely to exist. (Whittaker 2009) Exploratory testing is a useful way of testing in areas where scripted testing does not fit well. Also in case of inadequate specifications and severe time pressure, it is a way of getting a brief outline of the product. (ISTQB 2011)

2.1.4 The “what” of testing

Knowing how to test raises the issue of what to test on the table. Test types can be divided into two different categories: functional and non-functional tests. Functional tests test what the system does whereas non-functional tests focus on how the system works. Both of these two test types can be executed on all test levels. (ISTQB 2011) The focus of functional tests is on the external behavior of system features. Therefore, specification based black-box techniques may be used. (ISTQB 2011)

Non-functional tests test component attributes that are not related to functionality. Some types of non-functional tests are: performance, load, stress, reliability, usability, maintainability and portability tests. (ISTQB 2011) Performance and reliability tests are carried out to ensure that the software in general has no issues regarding computing power in general. No modules of the software must slow down the use of the software. (Kaner, Falk et al. 1999) Load and stress tests involve tests that are related to the maximum performance of the software. The software might be loaded with the maximum number of simultaneous users, or the most often used modules of the software might be executed in a fast loop. This is done in order to attain confidence that no defects arise when the software is under heavy load. Usability testing is usually carried out with non-expert users of the software to maximize independency in the use cases. Maintenance tests are run to ensure that the post-release updates are easy to upload since, typically, most money spent on a software development project is spent in the post-release updates. Porting is an activity which aims the software to run on a different operating system or computer. Port tests are used to check that everything goes smoothly in the porting process. (Kaner, Falk et al. 1999)

2.2 The generic testing process

As any development activity, testing is also a process. The process model used can differ between projects but some general guidelines can be drawn. Anne Hass (Hass 2008) formulates the generic ISTQB testing process as a three-phased process as shown in Figure 3.

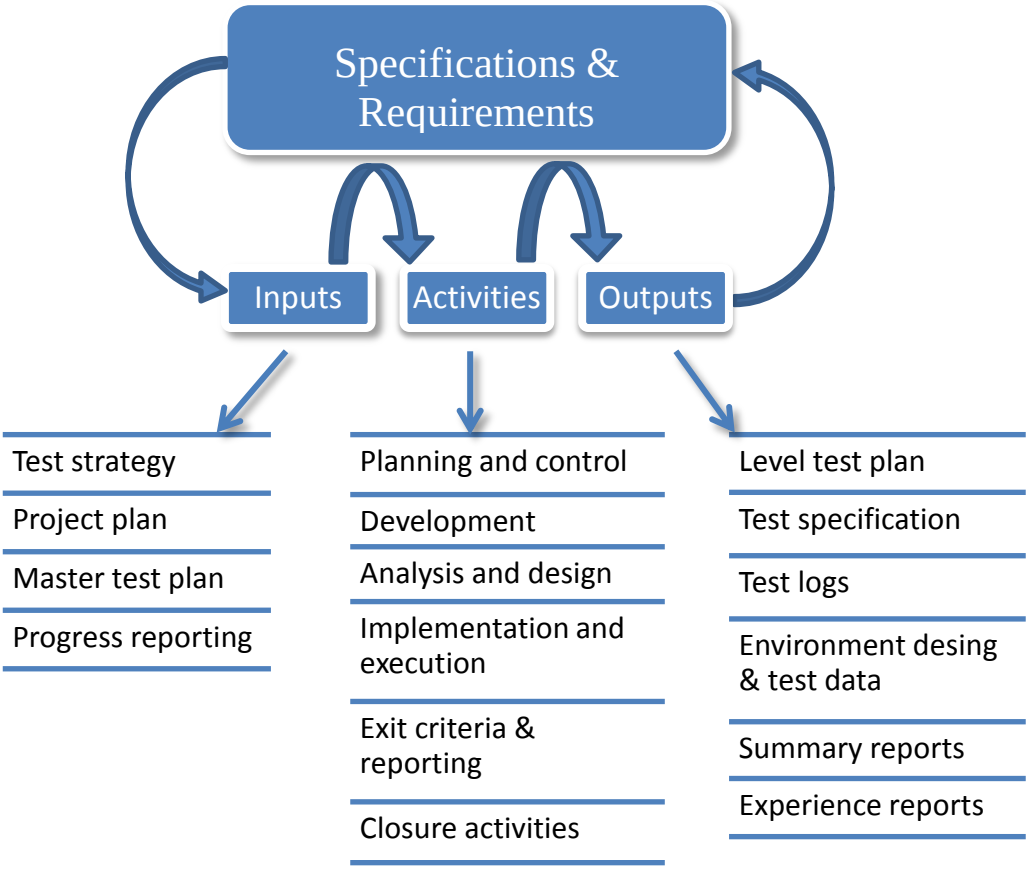

FIGURE 3. THE GENERIC TESTING PROCESS FLOW MODIFIED FROM ANNE HASS'S (2008) VISION OF THE ISTQB TESTING PROCESS.

The basic idea in the model is that in testing activities generating outputs are based on inputs defined in the requirement specifications. Furthermore, the outputs can generate specifications that can be used as inputs for some other activities. Therefore, the model is iterative as activities may have dependencies. Apart from the dependencies and some specific activities like test closure activities, the activities do not need to be executed in a strict sequential order. Activities like test control and planning in particular should be kept constantly in mind rather than executed once in the beginning of the test assignment. Even with very well-defined models, it should be borne in mind that they cannot tell us any absolute truths, as they are only models. All models should be tailored to the process at hand. (Hass 2008)

The subsections of the model are defined in the ISTQB Syllabus in the following manner (ISTQB 2011):

- **Test strategy** consists of analyzing the approach to the testing process i.e. what is to be tested. The approach can be anything, for example tool-, methodology-, analytical-, model- or consultation-based approaches. The selected approach depends on the product and test level at hand. An example of analytical approaches is risk-based testing where testing is directed to the areas with the greatest risk.
- **Project plan** is a document of the whole project – what is to be done, when and by whom
- **Master test plan** can consist of separate test plans for different testing levels. It is the key guideline for the testing. It can include descriptions of everything that

is related to testing such as testing scope, risks, scheduling, metrics selection and resourcing.

- **Reports** include all kinds of reporting inside testing and communication between necessary interest groups. Not only are the reports used for communication, but also for testing enhancements resulting from gained knowledge.
- **Test planning** is done to evaluate the objectives of testing and to make decisions that transform the test strategy into an executable ensemble. It is the responsibility of the **test control** to monitor that planned activities are on track.
- Based on the of the level test plan, **Test analysis and design** are performed to create high-level test cases and to specify the test environment.
- The procedures and/or scripts of the actual test cases are created in **Test implementation.** Finally, the actual testing part is done when the generated test cases are **executed**.
- During **Test execution** the **Test exit criteria** is constantly evaluated. The exit criteria is defined in the test plan and gives a guideline for the conditions that have to be met before the testing can end. The exit criteria is usually based on some metrics.
- **Test closure activities** are some of the few activities that have a specified slot in the end of the testing process. Primarily, they consist of gathering data, testware and experiences from the preceding test phases.

Compared to the ISTBQ Syllabus's process model, the advantage of the model presented is that it is an iterative process because no real life processes are simple straightforward processes. The concept of straightforward processes (of any kind) is commonly described in literature but rarely exist in the real world. In the world of testing, experiences have shown that at least three iterations have to be completed before testing can be considered complete (Hass 2008). Particularly the content of the steps provided in the previous model follows the Syllabus's steps respectively. Only the use and arrangement of the steps have been modified.

The generic testing process is always modified to the project's needs. The project at hand is not the only aspect affecting the testing process. Corporations may have their own testing policies that also affect testing. The corporate testing policy is usually a short high-level document on "why" testing is done in the company. It is usually developed by senior managers (including someone with experience in testing). The contents of a corporate testing policy usually include the business value of testing, a typical testing process, test metrics to be used and other aspects affecting testing. The policy is written to consolidate testing within a company. (RBCS 2012)

2.3 Test automation

When software is used to test software, it is called test automation. Automated tests are scripted via a framework and run whenever needed. (Kaner, Bach et al. 2002) Depending on the definition of testing, some might say that test automation is not testing at all. Since computers can only verify the software's response to match definitions, test automation can be considered checking of facts rather than testing. (Grahrai 2014)

2.3.1 Why automation?

As for testing in general, the main objective of test automation is to increase the reliability of the software with less time and cost (O'Leary 2008). Whereas a tester can

perform tests only for a limited time of the day with limited speed, automated tests can be run throughout the day without any need for breaks (Balasubramaniam 2014). Speed is a priority in testing, since the earlier the defects are detected, the cheaper and easier it is to fix them (Kaner, Bach et al. 2002). Vijay Balasubramaniam (2014) has written of some advances in test automation, greatest of them being:

- Fast execution of test scripts since schedules and resources will most likely not limit the performance.
- Manual reproduction of known defects is not needed since they are discovered automatically.
- Manual tests require contribution every time executed, whereas automated tests are reusable once scripted.
- Automated test are more reliable than manual tests if scripted properly.

On top of these advantages, some test types can be added to the list of the advantages of automation, since it is not feasible to execute them manually. Kaner et al. (2002) derive some of these being

- **Load tests**. How will a system allowing multiple users' simultaneous use react to the concurrent access of hundreds of thousands of users? In most cases, simulation is the only viable option to test great loads.
- **Endurance testing**. Memory leaks, stack corruption and wild pointers are common problems in software. Typically, they might not be apparent when they occur, but in the long run they can cause critical failure of the software. These problems might need days or weeks of uninterrupted use to appear.
- **Combination errors**. Some defects occur only when multiple functionalities are in simultaneous use. As modern software are becoming increasingly complex with an increasing number of functionalities, the number of different combinations grow exponentially.

Especially in mobile environments, automation comes in handy. The number of different devices and operation systems is huge. As mobile products are often required to run on as many device-OS-browser combinations as possible, testing all of these combinations manually is troublesome. Designing automated tests that can be run on any combination saves a lot of time and effort. (Rand 2014)

Although all levels of testing can benefit from automated tests, the biggest profit is gained from unit and regression test phases (Balasubramaniam 2014). Since automated unit tests are typically produced by the developer during or before coding (ISTQB 2011), they will not be further discussed in this thesis. Most development teams these days use agile concepts in their processes. This creates a huge demand for regression tests. New functionalities are added and existing defects are fixed in new builds, so need for regression is evident. (Balasubramaniam 2014) Most of these tests are executed in each test run. Therefore, it is convenient to automate them since regular manual repetition is not cost-efficient. (Palani 2014) For fast feedback, it is often wise to automate smoke tests as well. Smoke tests are tests that briefly go through the core functionalities of the product. If any of the tests fail, there is no reason to release the update. Once the smoke tests pass, the product can be released for regression testing. (Kaner, Bach et al. 2002)

Automation is a vital tool in testing but it does not mean that automating everything would be a sure way to success. Automation saves money and time in many cases but there are many things that cannot be automated. Exploratory and usability testing are good examples of tests that cannot be automated. Is the product easy to use? How does

it feel? Computers perform weakly in answering qualitative questions. They can only do what they are asked, and nothing more. (Balasubramaniam 2014)

In regression sweeps, automated test cases are reusable. The problem with the reuse is the flexibility issue among automated cases. Computers are not flexible. Even a smallest change in the specifications may result in all test cases failing since the test cases could not adapt to changes. (Balasubramaniam 2014) To enable adaptiveness, test automation always requires someone to maintain the tests – and that someone costs money. (Grahrai 2014) People are flexible by nature for changes in the environment, which is a benefit of manual testing (Balasubramaniam 2014). Hence, test automation should not be applied in applications under constant change. Still, the biggest reason for passing over test automation is the initial costs. Before even the first tests can be automated, it might take from months up to a year and huge investments in frameworks, environments and labor, whereas manual tests can usually be conducted almost immediately. Also for creating automated tests, the tester needs to have coding skills which might slow down the process. (Rand 2014)

Whether to automate the tests or not is a question that a simple return of investment (ROI) calculation can solve. Possible calculation parameters are scripting and execution costs for both manual and automated tests and a number of execution cycles. The savings in time and gained knowledge are a bonus for automation's merit. (Rand 2014)

Prior to automation, the test run should be designed properly. To begin automating without a proper plan can easily lead to the trap of automating things that are easiest to automate but poor in finding defects. The design of automated test cases differs a bit from manual case design. It is crucial to design automated cases independently so that the automation can be used to its full potential. Trying to automate existing manual cases easily leads to underutilization of automation. (Kaner, Bach et al. 2002) Cem Karner et al. (2002) summarize the message of this paragraph as follows:

"*Automating without good test design may result in a lot of activity, but little value.*"

This lesson should be kept in mind when designing automated test cases. Considering the organizations that lack test automation, all hope is not lost. The transition towards powerful use of automation can be done bit by bit by evaluating the core functionalities that should be automated and by writing the automated tests little by little. The capability and scale of the automation will then increase over time. When the automation of a certain functionality is at a phase where they it covers the existing manual tests, the manual tests can be given up. (Kuehlmann 2014) Yet, the decrease in time is not relative to the time the manual tests took. Test automation always requires maintenance. The more extensive the automation run is, the more maintenance is required.

2.3.2 Taking automation to the next level

Not only can the test case execution be automated, but the generation of the test cases can be automated as-well. One methodology aiming towards automatic test case creation is called model-based testing (MBT) (Blackburn, Busser et al. 2004). There are multiple model-based approaches, but the common thread of them all is that they create a hierarchical model of the system under test (SUT). The model's level of abstraction can vary a lot depending on the modeler, so even a non-technical domain expert might be able to use the model. The test cases are created automatically in this model using special MBT tools. These tools are further used to execute the tests. (Utting, Legeard

2007) The benefit of MBT is obvious – it requires less manual work. (Aho, Kanstén et al. 2014)

Typically in MBT the models need to be created manually. The second step towards even higher level of automation is to automate the modelling as well. GUITAR (GUI Testing Framework) is an example of a process model which uses graphical user interface (GUI) ripping tools to generate the models automatically. A GUI is the path with which the user communicates with the software. Memon et al. (2013) use the following definition for GUI:

"A Graphical User Interface is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events, from a fixed set of events and produces deterministic graphical output. A GUI contains graphical widgets; each widget has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI."

GUI ripping tools are generally crawler-like tools that do reverse engineering on the graphical user interface. For example, GUI Ripper is a tool that tries to log both visible and hidden GUI elements into a XML model called GUI Tree. Based on this hierarchical model, the automated test cases are created automatically. (Memon, Banerjee et al. 2013) These ripping tools have also landed in the mobile industry as Android apps have already been successfully ripped (Amalfitano, Fasolino et al. 2012).

However, not all projects can rely on MBT. Firstly, the methodology is highly dependent on tools and the tools are highly dependent on used scripting languages. (Aho, Kanstén et al. 2014) Secondly, the expected results of the tests (also known as Test oracles) are hard, though not impossible, to implement on the automated process (Blackburn, Busser et al. 2004). The third issue with the methodology is that it is hard to narrow down the automation scope. In complex systems, the generated amount of test cases is huge and most of these test cases test only very minor things or nothing at all. As a solution to this problem, test optimization tools have been developed which even further increase the tool dependency. Still, the biggest issue in automated model based testing is that the test coverage cannot be guaranteed as everything happens automatically. (Aho, Kanstén et al. 2014)

One way to utilize the methodology of the GUITAR process is to harness it in regression testing. Aho and Suarez (2014) propose the process not to be used primarily to find defects but rather to point out changes in the software. Whenever a new version of a software is released, a GUI model is created. The model is compared to the model of the previous version in order to locate any changes in the GUI architecture. This methodology enables tremendously fast feedback in development. (Suarez, Aho 2014)

2.4 Testing going mobile

Mobile apps can be divided into three different types which are Native, Hybrid and Web Apps. Native applications reside in the device and are accessible at any time. They have the power to use all the functionalities built in to the device such as camera and different types of sensors. Hybrid apps are like native apps with the difference that they are third party developed applications that run on the OS. The third group is called web based applications which are accessed via the device browser – native or hybrid. (Kumar 2011)

As discussed earlier, mobile software testing is a branch known to face tremendous challenges due to huge variety of different devices. In August 2014, there were almost 19.000 different Android devices alone (OpenSignal 2014). When taking into account

that all of these devices can be run on any of the dozens of Android versions, the number of device-OS-combinations is vast though many of the combinations are alike. This is where one of the key principles of testing needs to be revisited: Testing everything is simply impossible (Grahrai 2014).

As there are different types of testing for desktop software, mobile software have different test types as well. Narayanan Palani (2014) dictates eight different types of mobile software testing, some of which have a similar counterpart in desktop software testing. These types are:

- **Mobile Functionality, Performance, Usability, Security and Compatibility testing** are test types that have their counterparts in desktop software testing respectively as described earlier. These test types have no major differences compared to their desktop counterparts. (Kaner, Falk et al. 1999, Palani 2014)
- In **Mobile Interrupt Testing** the use of the application is interrupted by normal smartphone functions like incoming text messages and calls or push notifications. Pressing all hardware buttons on the phone and checking the app's response is also considered interrupt testing. (Knott 2014)
- **Mobile Interoperability Testing** refers to testing that is done to ensure that an application will work through mobile communication networks. Mobile devices are used in environments where various networks using different technologies coexist. The crosstalk and interference of these networks can affect the behavior of the software. (Álvarez 2012)
- **Mobile Localization Testing** is a phase of testing that ensures that the product matches the expectations of the cultural characteristics of the target countries. Most common type of localization testing is the linguistic verification of the supported languages but other cultural aspects should be checked as well. Software may carry cultural assumptions that some cultures may see as alien or hostile which can result in a severe flop of the software in the market area. (Keniston 1997)

All of these test types are important but the prioritization depends naturally on the project at hand.

There are a couple of ways of doing manual and especially automated mobile testing. Figure 3 illustrates the four different levels of virtualization that are browser add-on based automation, simulator & emulator based automation, remote device automation using cloud services and real local device automation using bots. Each of these levels are applicable at some phase of the testing depending of the project at hand. Each one of them has its own advantages and disadvantages. The lower you descend the pyramid, the more reliable information is gained in terms of quality assurance. In proportion to the better knowledge comes the cost of automating tests. Real local devices are the most difficult to automate resulting in higher cost. (Sriramulu, Ramasamy et al. 2014)

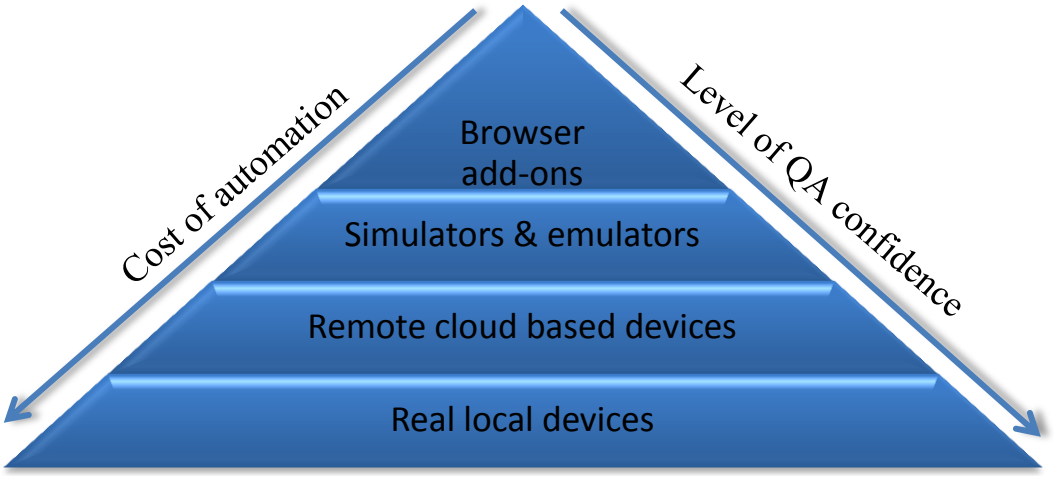

FIGURE 3. DIFFERENT LEVELS OF VIRTUALIZATION IN DOING TEST AUTOMATION ON MOBILE DEVICES.

Browser add-ons are features in desktop browsers that open web pages mimicking the look and feel of a desired mobile device. Therefore, only web-based mobile apps can be tested with these add-ons. At least Mozilla Firefox and Google Chrome browsers have this add-on (Buonamico 2014). Browser add-ons are by far the easiest and also cheapest way of doing mobile automation. This is due to the possibility of utilizing desktop automation software. There are multiple open source automation tools and frameworks in the market free for use (Sriramulu, Ramasamy et al. 2014). One of the most common open source tools is Robot Framework which is a generic keyword based framework that uses different automation tools such as QTP or Selenium simultaneously (Neal 2013). Since the add-ons make the web page only look real, only functional testing can be done using add-ons. Issues rising from screen resolutions, performance parameters and general device compatibility cannot be tested. The use of browser add-ons can gain only a fairly low level of QA confidence. (Sriramulu, Ramasamy et al. 2014)

For a better level of confidence, simulators or emulators can be used. In this thesis, software that simulate the behavior of the mobile device and that can be run on normal PCs are called simulators. In some studies, the word simulator may also refer to browser add-ons (Rayachoti 2012). Mobile phone manufacturers put a lot of effort in developing these simulators to ensure their platforms are easy to test. This results in the generally high quality of these simulators. Simulators are available for all operating systems and a big variety of different devices. (Sriramulu, Ramasamy et al. 2014) Most of the emulators are free of charge to download and use (Palani 2014). From a developer's point of view, one great benefit of emulators is the access to processes that lie beneath the screen of the physical device. This makes the debugging process easier. (Rayachoti 2012) In addition to the tests, browser add-ons also enable interruption, and device-specific characteristics can be tested via simulators. Although simulators enable more testing than browser add-ons, they do not replace real device testing. The simulators only attempt to simulate the real devices – they cannot ever be real devices themselves. Firstly, when a defect is found on a simulator, we cannot know for sure whether it is a real issue in the software or a just simulation error. (Sriramulu, Ramasamy et al. 2014) Performance is the second problem with using simulators. Mobile gadgets come with limited CPU and memory compared to a desktop computer. The latency occurring in real devices needs to be taken into account when designing simulator tests. (Palani 2014)

The first step towards real device automation is using cloud-based services. There are multiple service providers in the market that offer a huge capacity of different mobile devices to be used in the cloud for automation. In these services, the scripted test cases are uploaded to the server of the service provider where the tests are executed on real devices. Usually these services come with monitoring tools to ease the analysis of the results. With this kind of automation, almost every test type can be executed. These cloud services provide two major advantages. For mobile products that are targeted to be run on as many different devices as possible, these services provide testing equipment. If a product is to be tested on dozens or hundreds of different devices, the procurement cost of these devices will be huge. The second advantage is the possibility to do automation from anywhere. In multinational companies the testing team and the development team might be working in different locations, or there might even be multiple different testing teams operating on the same product, it is vital that all teams have access to the devices. The disadvantage of these services is the high licensing costs. Even the pay-as-you-use services can become pricy. Although this is fairly close to real device testing, it is not exactly like real device testing. (Sriramulu, Ramasamy et al. 2014)

The only way of doing proper real device testing is to use local devices for automation. This can be done in a couple of ways. One way for companies is to create a local lab of devices, as in the cloud service version, and maintain the service themselves. This might also be considered a cloud service. In these cases the devices are accessed via USB or WLAN. Depending on the framework and devices used, this method might require rooting or jailbreaking the devices. (Sriramulu, Ramasamy et al. 2014) Rooting and jailbreaking are activities that are done to Android and iOS devices respectively to gain superuser access to the devices. The superuser access allows the user to have more rights on the device than a normal user would. The reasons behind rooting and jailbreaking vary but one of the most relevant reasons is to run apps on the devices that are not allowed by the OS providers. These activities may create security breaches because the increased rights are not limited to the desired apps as the superuser rights apply on all apps. (Vidas 2011, Garner 2011) Modifying the device to a state that is prohibited by the manufacturer and that is not common practice among the users has weak results in terms of QA confidence. Instead of a local lab, a second approach to real device testing is to create a physical robot around the device. Generally these creations are robots that tap the screen in desired locations. (Sriramulu, Ramasamy et al. 2014)

The question of what level of virtualization is needed depends on the testing scope. A rough guideline is that it depends on the state of the project. The earlier the state of the process, the more virtualization can be used. When performing basic unit tests for a new app, browser add-on or simulator tests are most likely more than enough. When moving towards a more complete product, less virtualization is to be used. (Sriramulu, Ramasamy et al. 2014)

2.5 Choosing the right automation framework

Test automation is not a process that can be started from scratch without any preparation. Before scripting can start, the testing environments and, most of all, an automation framework needs to be set up. Prior to the set up, the framework needs to be chosen. The question is, which one to choose as there exists many different automation frameworks with different purposes?

An automation framework is a communication gateway between the tester (or the coder creating the scripts) and the system under test. The automation scripts are created via the framework and executed whenever needed.

A key factor in mobile test automation is the vast diversity of devices to be tested. (Narasimha 2013) For example, if a mobile app is required to be run on most Android machines, tests can be run on dozens of different devices. The amount of repetitive work is overwhelming and is an issue that automation can solve (Narasimha 2013). When choosing the automation framework, a good starting point is to go through the parts that are in the need of automation. All automation tools have their own pros and cons. The problem is that there rarely exists an ideal test automation tool which complicates the decision-making. (Kumar 2012)

Script usability is the most critical point of automation tool selection. As the point of test automation is to lower test execution costs, there is no point in creating test scripts that can be only used on one device (the only exception being the app is targeted to run on one device only). This is why the framework should support all desired devices if possible. Lack of complete support over the desired operating systems results in the use of multiple frameworks, which eventually leads to increased costs. (Narasimha 2013, Sridharan 2014) In some cases, this ideology of one framework might not be the best solution as Govindasamy (2012) argues. He states that the optimization of the tool selection should be focused on proper platform support which in some cases can lead in the use of multiple tools. Depending on the scope of the automated tests, the tool should support both real devices and simulators since some tests are more convenient to be run on simulators as mentioned above. (Kumar 2012) Naturally, if an app is desired to be run only on a specific OS, an OS-specific testing framework most likely is the best alternative.

To ensure the reuse of scripts, Palani (2014) suggests the use of structured frameworks. Structured frameworks use object libraries from which they select a specific element to be used in a particular case (Palani 2014). It is a useful method, especially in mobile test automation's picture comparison, since the element at hand can be compared to a library of elements where an associate should be found. Combining this method with a data-driven and keyword-based automation framework results in highly reusable scripts. The principle in data driven test scripting is to run general scripts with parameters to execute a command. The greatest advantage of data driven testing is the reduced amount of keywords as commands that are alike are combined into one single command which is driven with different parameters to perform the wanted action. (Palani 2014) As a great advantage, keyword-based frameworks enable non-technical testers and business users to familiarize with test automation as keywords are easier to adopt than plain code (Neal 2013).

Test automation usually involves clicking on elements and browsing through the software. There are two different alternatives for this: either the automation is done based on the GUI elements or the native elements of the software such as "ID" or "name". (Palani 2014)

The first option proposed requires naturally a fixed GUI where graphical objects such as pictures can be contrasted. These objects can then be clicked, dragged or whatever a normal user would do with graphical objects. For testing purposes, image finding can be used to verify whether an image exists on the screen or not. (Zhifang, Bin et al. 2010) As users explore software in almost every case based on graphical elements visual on the screen, this method is very close to real user actions. As a downside, this method is highly dependent on the screen resolution as it is complicated for the computer to

associate two same pictures with different resolutions (Borjesson, Feldt 2012). In addition to the browsing of graphical elements, optical character recognition (OCR) can be used. OCR is a method used to extract text from a picture. OCR commands do not only return the text in a picture, but also the location of the text. (Zhifang, Bin et al. 2010) This is useful information if a button with a specific text is to be clicked and the button can be located anywhere on the screen. The downside of OCR is that the font and color of the text need to be clear and known in the software using OCR, and that the background needs to be light enough as well as solid to ensure the effectiveness of the OCR (Zhifang, Bin et al. 2010).

The second possibility to browse the software with automation tools is to use native object elements such as buttons, images, lists and other elements used by the application. When coding the software, the buttons have specific names or IDs. These tags can be used to manipulate the software. Object-based test tools enable automation by mapping the elements on the screen and creating a hierarchy where everything exists. (Narasimha 2013) This method requires that the source code enables these tags to be visible. At least in HTML-based web software this is a commonly used technique. (Palani 2014) Testing tools like Selenium 2 use this technique and are efficient in website and mobile app test automation (Buonamico 2014).

Although it is essential to select an automation framework that fits the project at hand as well as possible, there are still some factors that need to be considered before making any final decisions regarding the mobile automation framework. First of all, the licensing price of any automation framework can be high. Naturally, there also exists many open source solutions that are usually free for use, but these tools might not be updated as often to match the vastly changing environments. If one considers choosing an open source solution, the stability of the tool evolution needs to be verified. (Sridharan 2014) Secondly, if the automation project is carried out by an organization that already has done test automation, the ability to integrate the existing systems is a matter that ought to be taken into consideration. In most cases, implementing a completely new framework on top of other frameworks results in highly increased maintenance costs. (Grebenyuk 2013) A third point slightly relating to the previous bit is the programming language in use. A framework might support multiple scripting languages. Therefore, choosing a framework that supports a language the test staff is familiar with is usually more effective than adopting a completely new language (or any new technology) to the corporation. (Sridharan 2014) As learned before, the automated tests need to be maintained in all cases, and for obvious reasons relying on only one person's knowledge is highly risky business.

After a framework has been selected, a proof of concept on critical platforms should be conducted (Narasimha 2013). The purpose of a POC is to dispel any last doubts about the chosen framework before going ahead with the automation process (Grebenyuk 2013).

2.6 Mobile platform testing in RAY

2.6.1 RAY's testing principles

The basic function of the testing and quality assurance department of RAY is to provide software testing services for RAY's internal projects. Multiple testing tools are used to make the testing easier. In almost every system the main frame of testing is based on

use case test cases. Depending on the system under test, various levels of automation is utilized. Typically the core of manual testing is scripted test cases. Test cases that are specified in a high level of detail have multiple advantages. First of all, they leave no room for misunderstanding. Anyone can check afterwards, whether a specific functionality was tested or not. Secondly, highly detailed test cases do not depend on specific individuals. In case the person responsible for the testing of the system is absent, most likely any co-tester can perform the tests.

2.6.2 The component architecture

Playtech Ltd. is a major operator in the online gaming industry. Typically Playtech provides all necessary components for running online casino business to their clients. For such customers, Playtech deploys all production updates themselves. This means that there are online casino operators that offer the exact same product. Usually the only difference between the products is the graphical user interface which is tailored to the customer's needs. The tailoring operation is also known as skinning. Figure 4 illustrates two different skins of Playtech's downloadable Poker client. The left one is RAY's and the right one is William Hill's poker client. William Hill is also an online gaming company whose products are delivered by Playtech. The clients are alike, though there are some discernable differences in the elements and color choices.

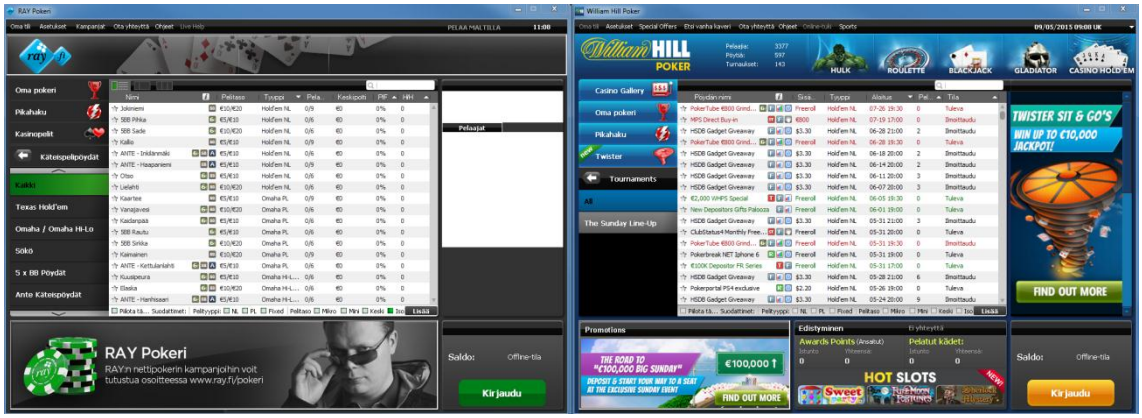

FIGURE 4 RAY'S AND WILLIAM HILL'S POKER CLIENTS (RAY 2015, WILLIAM HILL 2015)

In RAY's case, Playtech provides mostly the same components they provide other clients as well. The biggest difference is that Playtech's backing systems are integrated with RAY's own. RAY cannot completely rely on Playtech's backing systems since RAY's other operations, mainly slot machines, require their own systems. Due to the strict regulations under which RAY operates, the control over the production releases has also been stripped from Playtech. It is in RAY's interests to have complete control over their product.

As for all online gaming components, the mobile casino software RAY uses in their mobile games is also produced by Playtech. The mobile environment can be roughly divided into two different points of interest. These are the casino game client itself and the gaming server including the backing systems. The games come along with the game client, but all user information and gaming data, including balance changes and game round details, come from the gaming server. The architecture minimizes the possibility of misuse since all gaming related events such as lotteries take place on the server side. Furthermore, the game client is separated into two different platforms: MGP and NGM as was described earlier. The differences between the two are minimal from the user's

perspective as the user interfaces have only minor visual differences. Beneath the surface the platforms are completely different however.

All new games are released on the NGM platform. The MGP platform is kept alive since a great deal of the games run currently on the platform. Porting all games from one platform to another would require more effort than maintaining two separate platforms – at least from Playtech's point of view. Both gaming platforms and the server software update at regular intervals and all updates need to be tested separately.

2.6.3 Testing the already tested

As a recognized operator in the online gaming industry, Playtech naturally tests their products before shipping them to their customers. This process applies also to the mobile platform RAY uses. So one might ask, if Playtech tests the product prior to launch, why is testing needed in RAY in the first place?

There are more than one reasons for RAY's own testing. Firstly, testing is rational from a business point of view; it is cheaper to test the software than to deal with buggy software in production. As discussed earlier in this thesis, buggy software in production generates mistrust among customers and leads to direct and indirect loss of revenue. Secondly, RAY's interests differ a little from Playtech's interests. RAY operates in a highly regulated market which leaves little room for misconducts. Offending any part of the rules set by the Ministry of Interior (Ministry of the Interior 2013) give the Ministry the possibility for counter-measures against RAY – worst of them being complete rundown of the operation. The legislation under which most of Playtech's clients operate is not that strict. This means that Playtech can direct their main focus of testing efforts on other features than regulatory characteristics. Due to strict regulations, it is in RAY's interests that no regulations are offended. Also, it is notable that most of the regulatory features that come along with the software are specifically tailored for RAY. From Playtech's point of view, RAY is a small customer and RAY tailored features do not create a big business risk as a whole.

The third point is that though the games are provided by Playtech, the backing systems are made both by RAY and Playtech. RAY has their own backing system for their client databases and other operations. Playtech games run on their own backing systems. In practice, this means that these backing systems work simultaneously when games are played. The crucial part is the communication between these systems as the systems work separately. Despite of Playtech's testing efforts, problems may occur, since they have not tested the product against RAY's backing systems. Figure 5 sums up the development cycle taking place prior releasing a new build.

FIGURE 5 THEORETICAL FLOWCHART OF RAY'S NEW CASINO RELEASE

In general, testing performed by RAY's testing team for their mobile casino can be seen as one-phased acceptance testing. Playtech releases only final builds, never anything under development, for their customers. This applies to RAY as well. Hence, the product ought to be ready for production, i.e. RAY's tests are the last tests performed before production. Using quality information testing produces, RAY's business leads decide whether the build is a Go (ready to release) or No-Go (will not be released). In

other words, it can also be seen as integration testing since the integration between Playtech's and RAY's systems is being tested for the first and only time during the tests. The RAY tailored features complicate the ensemble. The features are tested against RAY's backing systems for the first time. From this point of view, the test phase contains also elements of system testing. As icing on the cake, the team performs regression tests to verify defects found in earlier versions of the software.

Namely, the testing that the team performs is user acceptance testing. Due to the reasons described in the previous chapters, testing performed in practice can be described as follows:

Something between integration and regression testing with a severe user acceptance touch.

2.6.4 Legal specifications concerning the mobile platform

The Ministry of Interior has dictated some responsible gaming rules in the Lotteries Act which affect all online gaming including the mobile platform. The rules are strict and violation of any responsible gaming features entitles the players straight compensation for their losses. The sections in the gaming permission that affect mobile gaming the most are (Ministry of the Interior 2013) the following:

- When registering to the casino for the first time, player must set daily and monthly loss limits. These loss limits represent the maximum amounts player can lose within a day or month respectively.
- Player is able to set an optional time rate for an individual gaming session.
- Casino games are divided into two different categories: slot games and table games. Within each category player can lose five hundred euros (500€) per day at maximum.
- Player can exclude him- or herself from all gaming activities. The maximum length of the exclusion is 12 months. The exclusion cannot be withdrawn within its validity.

In practice, the time rate involves two separate functionalities. Either the player's gaming session is automatically ended or the player is only notified of it when the specified maximal session time is reached. Also, two separate self-exclusion alternatives exist. Player can exclude him- or herself from poker and casino gaming separately.

The gaming permission dictates also some other restrictions concerning the mobile platform such as deposit limits. (Ministry of the Interior 2013) Testing these functionalities have been differentiated from the liabilities of the internet gaming team of RAY and therefore are not discussed further in this thesis.

3 Results

The results of this thesis will be studied in this chapter. They are separated into three different sections: The Test Plan, Framework Selection and Proof of Concept.

The order of the testing process is very strict and thus this chapter will follow the same order. The test plan must be derived prior to selecting any automation framework. The concept was introduced already in Chapter 2.5 based on Kumar Pradeep's thoughts.

3.1 The test plan

The first objective of this work was to create a viable test plan for the mobile casino software. As learned in the previous chapter, the ISTQB syllabus determines that a well-defined test plan contains not only the tests to be executed but also a proper description of the test environment and test prioritization among other aspects.

As described earlier in this thesis, the main purpose of testing in any company is to provide quality-related information for the business representatives so that they can make justified decisions on the product. These business representatives and multiple experts in RAY were interviewed in order to gain knowledge on what information they wish after each regression run. One clear message recurred in all the conversations: focus should be set on the functionalities, especially on those interacting with any RAY integration or RAY tailored features. All in all, the role of non-functional testing in the project was somewhat complicated. RAY does not possess much power to influence Playtech's development. In practice, RAY has only minimal access to influence any usability or gameplay related features. Therefore, it does not make sense to focus testing efforts on areas that are least likely to ever be developed.

The key factor in creating the new test plan was not to place any focus on the fact that one aim of the project was also to create test automation for the system. The point was to concentrate on what was wanted and needed for the mobile software quality assurance. By taking this approach, the pitfall of creating an automation driven test plan was avoided. If the focus of the test plan had been automation, there would have been a great risk of missing important test subjects. In the end, the main goal of the project was proper and efficient quality assurance – not creating a cool automation without proper purpose.

Since RAY's QA team has to execute all tests without any knowledge of the games' source code, the test techniques used are primarily all black-box techniques. The tests are all designed based on the expected behavior of the system. In a way, it can be said that also gray-box techniques are utilized as knowledge of component integration is used in testing. Although the source code of no component is known, the functionality of each component is known. Thus, some tests can be simplified based on the assumptions of the component integration.

The detailed test plan is provided in Appendix 1. It consists of 57 use case tests. In the new test plan, test cases can be divided into six categories based on functionality or test type:

- **Information security tests** consist of any login or user detail related tests.
- **State transition tests** verify the website's functionality as every path has to have a correct destination.
- **Gameplay tests** verify all balance and history logging related issues and that the games are playable.

- All **responsible gaming** features are tested to ensure the legal exigencies and implications of the triggered RG features.
- Minor focus is also set on **performance and interruption tests**. As interruption tests require active use of other apps, the tests are performed to gain knowledge on what can happen when using different popular apps simultaneously. The main purpose is not to find defects in the system under test as the problems occurring might root in other apps.
- **Exploratory testing** shall also be performed on the system. The main focus of exploratory testing is the general functionality of the games and visual appearance. This is separated as its own test case since the outcome of any played gaming round is impossible to dictate because the results of every round are completely random. As a minor part of exploratory testing, the look and feel of the software shall also be tested.

The test plan is based on the following assumptions and reasonings:

- As the game clients are developed more or less separately for all three supported operating systems (iOS, Android and Windows Phone), they need to be tested separately as well. Experience has shown that most defects are platform-dependent. Thus, it is even more important to test all supported operating systems separately. Within the operating system, the choice of device is free as long as it is a supported device. To find device related defects, mostly having to do with imaging, alternating between different devices is recommended.
- All gameplay or responsible gaming features are tested on both gaming platforms (NGM and MGP). The platforms are developed separately, and therefore are prone to platform dependent defects.
- If a certain functionality works in one game, it is assumed that it works in all games on that platform. This applies to all responsibility tools, gameplay features and state transitions. All games are tested thoroughly prior to release. New builds of the software rarely have any effect on the games as they focus on updating the platforms. If a build is known to have fixes in some specific game, the testing efforts are aimed on that game. Although the games are not supposed to have any changes, it does not mean new defects could not arise within new builds. Yet, this simplification is essential to keep the extent of the test run manageable. As mentioned back in Chapter 2.4, testing everything is impossible.
- If a functionality works with one valid parameter value, it works with every other valid parameter value possible. It is impossible to test every possible parameter value if the parameter is an integer without boundaries.
- All self-exclusion related tests have the precondition that the self-exclusion is set on. A second palette of tests is those that activate the self-exclusion and verify that the activation was successful. The distinction is made in order, once more, to control the amount and length of test cases. Although self-exclusions with varying lengths are activated only via the casino software's desktop version, a 12-hour panic button is present virtually in every gaming view; not only in the desktop version but also in the mobile client. Activating this panic button excludes the player from all gaming activities and thus is one form of self-exclusions.

The assumptions and simplifications are vital to the testing process. Resources available for testing the mobile platform after each update are limited and need to be allocated

properly. Although increasing resources would slightly step up the general reliability over the build, the profit would not be significant.

RAY provides two different environments for online game testing: A system testing environment, and an acceptance testing environment also known as staging environment. The difference between these two environments is the possibility to simulate events. The system testing environment allows certain things, such as balance changes and the Finnish Population Register Centre, to be simulated, whereas the acceptance testing environment is a replica of the production environment. To be precise, the production environment RAY uses is a replica of the staging environment.

As was shown in Chapter 2.1.2, the closer the testing events take place to production, the closer the testing environment should be to the production environment. Though some particular tests would be easier to execute and especially to automate in the system testing environment, the staging environment is chosen as the test environment in this test plan. The tests are the last tests executed prior to production, and therefore event simulation cannot be seen as a potential way of performing the tests in this case. The decision was made due to the general fact that tests performed in the staging environment provide better quality-related information.

The tests are prioritized into four different levels, P1 being the highest priority level and P4 containing the lowest priority test cases. Test prioritization is done to ensure that test cases are carried out in the right order. A certain prioritization level is given to each test based on its business impact. When testing efforts begin with the highest level cases, the most critical defects are more likely to be found early. Priority one test cases are all such cases in which failure of any single one of them would lead to immediate rejection of the build. Priority two cases consist of tests that would most likely result in the testing team's strong suggestion not to release the build if any of them should fail. Although the level three and four test cases test less business-critical aspects than the first two, they are still important cases.

The plan, as described in the Appendix 1, contains not only the test case titles but also few sentence descriptions of the motives and targets of each case. To honor RAY's testing policy, all test cases are scripted in one of the QA team's testing tools regardless of whether the tests are manual or automated. An example of a scripted manual test is provided in Table 1.

TABLE 1 AN EXAMPLE OF A SCRIPTED MANUAL TEST

Title: Gameplay – History – Data		Priority: P1
Preconditions: Tester has a valid gaming account and access to the backend system		
Steps		Expected results
1.	Open RAY mobile casino web site	The web site opens fluently
2.	Log in to the system	Log in is successful
3.	Open any MGP game	Game loads

4.	Play one round – Write down the game name, date and time, balance prior the game round, bet distribution, total bet, win distribution, total win and balance after the game round	One round is played and the round details are written down
5.	Go to RAY's desktop web site -> My account -> Gaming history -> Casino history	History tool opens
6.	Search the game round played for history information and compare the details to the notes taken on step 4	The details provided in the history tool correspond precisely actual events
7.	Log in to the backend system and search for the played round's details and compare to the notes taken on step 4	The details provided in the backend tool correspond precisely actual events
8.	Repeat steps 3-7 for any NGM game	See expected results of steps 3-7

It bears mentioning that, some responsible gaming tests, such as the cases testing the daily loss limit, are divided into multiple different tests. For example the priori one case, *Responsible gaming – Gaming limits – Daily loss limit unexceedable*, tests that the daily loss limit cannot be exceeded. This test does not take into account what happens after the player has tried to place a bet that does not fit in the daily loss limit. As long as the loss limit exceeding bets cannot be placed, this test passes because the legal prerequisites are fulfilled. The second priority test case, *Responsible gaming – Gaming limits – Daily loss limit about to exceed*, focuses on the boundary values and that the software operates correctly when the loss limits are reached: the software does not crash, the player is notified with the correct pop-ups, the player can continue gaming with a smaller bet etc. Although failure in the latter case is highly unwanted, it does not create a statutory threat to business. The same differentiation is made with all responsible gaming tests to make a clear difference between the statutory and non-statutory tests. In case the execution time for the test run is highly limited or only a brief smoke test is wanted, the test run can be completed by running only the first case. Although this option exsits, it is not recommended to carry out the test run without the latter test case because it risks the reliability of the build.

In addition to the test prioritization, each test case was given an automation prioritization level, A1 being the highest and A4 the lowest automation priority. When deciding on the level of each case, the factors that affected the decision were:

- The effect on other cases to be automated i.e. if the case is not automated, will other cases be impossible to automate?
- The general prioritization level which is related to the business value of the case.
- An experience based evaluation of the automation difficulty of the case. If multiple cases have the exact same automation value, automation ought to start from the one that is easiest to automate. The easier the case is to automate, the faster it is developed, and therefore the faster it can be set into operation.
- The difficulty of performing the test manually.

The cases with A1 prioritization level include functionalities of which almost every test case is dependent on: logging in and opening the games. Therefore, they are the ones

that ought to be automated right in the beginning. It should be mentioned that almost every login related test case was tagged with the A1 priority, even though only the function of logging in with a real account is relevant to other cases. The reason behind this relies in the ease of the automation as the only major differences are the parameters used. Otherwise they are alike.

Together with the A1 cases, the second priority A2 cases form the core automation scope of the project. The A2 cases consist of the most business-critical tests that most likely are easy enough from the automation perspective. As in the A1 class, also the A2 class includes tests that have minor business value but sponge with similar and more relevant cases, and thus are automated early in the project.

The A3 and A4 level test cases have either less business value or are fairly complex from the automation point of view. The possibility and need for automating these cases is revised later as the A1 and A2 level cases are automated. If a case is complex, resulting in high development and maintenance costs, the benefit of automation is lost. Some of the lower automation priority level cases can be partly automated though, if only a particular step creates difficulties.

Some cases were tagged without any automation priority. Automation does not work well with these cases by any level and they need to be executed manually. Most of these cases include a certain degree of abstraction which is always problematic for automation purposes. These cases usually lean towards exploratory testing. As already noted in Chapter 2.1.4, exploratory testing and automation do not sit well together. A good example of abstraction in the test plan is the test case number 20: *Information security – Lobby – Games not reachable without logging in* in the test plan. This test case is extremely relevant since the games are not supposed to be reached without logging in. The problem with the test case formulation is that no one knows how the games could be reached without logging in. Finding the right way to abuse the system relies on the tester's experience. When the way cannot be specified, it is impossible to automate the test.

3.2 Test automation framework

3.2.1 General comments

In the test plan discussed in the previous chapter, it was shown that a great deal of the test cases can be automated. As it was shown in Chapter 2.3.1, the reasons for test automation are evident. Fast and reliable execution of tests on multiple platforms after each update gives a general view of the build's state. Although automation is harnessed, there are no needs for automation-specific tests, such as load or endurance tests, due to the reasons dictated earlier. In this case, this does not diminish the need for automation as the needs focus on other aspects.

In addition to the aforementioned advantages there is one more major benefit that comes to support automation. In the test plan it was assumed that only an update of the game client affects mobile gaming i.e. heavy tests are conducted on the mobile platform only after new client builds. RAY's online gaming architecture consists of dozens of components that are all separately updated. In most cases, an update of a single component does not affect the others in any way. In reality, mistakes do happen and other components can be affected unintentionally. Altogether, component updates are almost daily events. Although it would be nice to test all products after any update, it would result in everybody testing everything all the time. This would not be cost efficient – at least if performing the tests manually. This is where automation comes in.

Computers do not work according to any regular office hours or require coffee breaks every now and then. With proper automation, it is possible to run tests on the most business-critical parts all the time, even when no manual tests are to be executed. This way, information of any critical change in the product is disseminated almost immediately after any update.

The pros and cons of utilizing test automation in RAY's case are listed in Figure 6. As long as the balance beam between pros and cons is greener than it is red, test automation should be considered a viable option. In this case, the main reason for adopting test automation in RAY's mobile environment is the belief of reducing manual regression testing costs.

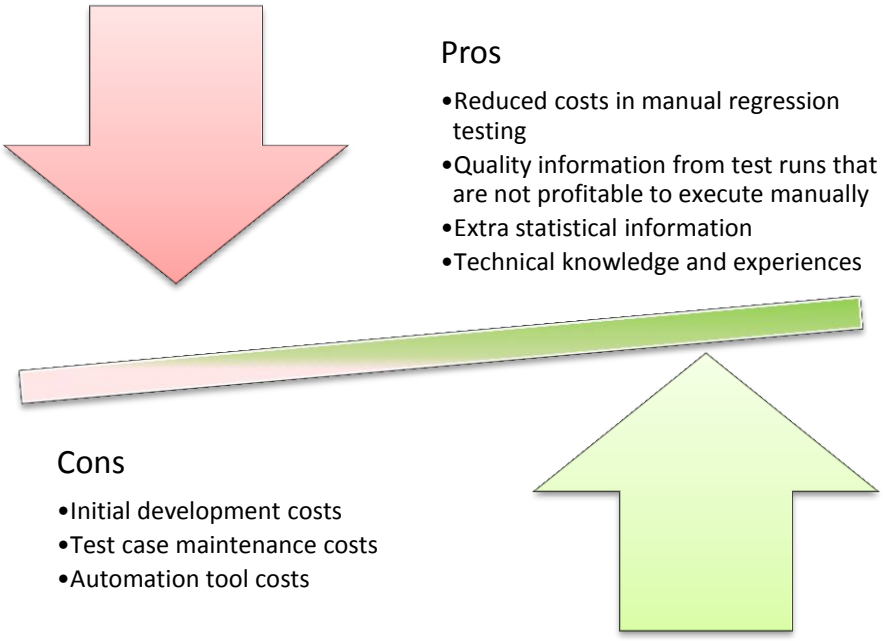

FIGURE 6. PROS AND CONS OF MOBILE TEST AUTOMATION IN RAY'S QA.

To get the very best out of automation, a proper framework needs to be chosen. Multiple different options exist, each of them having their own pros and cons. Both open source and commercial tools are available. So which one of them to choose?

For other automation purposes on desktop machines and products, Robot Framework is the main automation tool in RAY's QA. Robot Framework is an open source keyword-driven acceptance testing tool. The syntax it utilizes is based on Python. As an extension to the Robot Framework native libraries, Python and Sikuli scripts and in-house developed libraries are also used. Sikuli is also an open source tool used for image based GUI automation. All automation tests are developed into a continuous integration (CI) environment i.e. whenever a single test case is developed or updated, it is set off to work immediately. To manage the CI environment, an open source CI server, Jenkins, is used. According to Mithun Sridharan as mentioned in Chapter 2.5, it would be ideal if the chosen framework would integrate some of these components, since it is simpler to expand existing technologies than to adopt completely new ones.

In practice, the level of virtualization has the biggest influence on the chosen framework. As discussed in Chapter 2.4, the lower the level virtualization, the better the QA confidence. Unfortunately, it comes with an increased price tag as the automation technology becomes more complex and thus more expensive. Currently, all (manual)

mobile testing in RAY is done on real devices. It is also common knowledge that Playtech tests their mobile products solely on browser add-ons. Doing this, they can easily ensure a higher capacity of supported devices. On the other hand, this is considered the primary reason why RAY's mobile QA has been so far so successful and efficient in finding defects in the products. The use of real devices provides the best possible quality confidence. As stated in the previous chapter, testing efforts focus on the functionalities of the software. According to the findings introduced in the methodology section, it would be enough to execute most of the tests, especially those that are to be automated, on simulators or emulators. After having multiple conversations with RAY's business representatives and system specialists, a consensus was drawn that test automation should focus on real local devices if possible. The reason behind the decision is that RAY wants to keep up the high standard of quality assurance despite the increased price. Therefore, the framework candidates ought to have real device support.

Based on the knowledge gathered in the previous sections, the following criteria were used in evaluating different framework solutions:

- **Real device support.** As discussed, the focus ought to be on real devices, not on emulators.
- **Supported mobile platforms**. An ideal framework supports all three major operating systems (iOS, Android and Windows Phone). If the support does not cover all OSs, value is given to each supported OS based on the number of RAY's mobile players using that specific OS.
- **Supported mobile browsers**. To have appropriate support for a specific OS, the framework must enable the use of any officially supported browser that are on Android Chrome and native browser, on iOS Safari and Chrome, and on Windows Phone Internet Explorer. This requirement is to ensure that tests are executed in an environment that is popular among users. Testing in an environment that has minimal use does not give reliable quality related information.
- **Maintenance costs.** In practice, it is required that the framework is either somehow keyword-driven or supports some kind of function calls. It is essential for easing maintenance efforts. In the end, the number of automatable test cases is in the hundreds. E.g. a change in the login feature in the game client takes place, it is easier to make the necessary changes into one login function which every test case calls rather than updating hundreds of different test cases.
- **Usability.** The easier the framework is to learn and use even for non-technical testers, the better. In the future, there might be people that are not dedicated to the framework developing tests to the mobile platform. Also, it is faster (and therefore cheaper) to develop and maintain test cases on easy-to-use frameworks.
- **Integration to RAY's existing systems**.
- **Price tag** i.e. open source vs. commercial tools. Although open source tools are free for use, stability and future development activities can cause issues and, hence, must be verified.
- **Need for rooting or jailbreaking devices.** Some technologies require extreme measures for them to run. For better quality knowledge, these actions are not recommended.
- **Use of the technology outside this project.** RAY has other projects operating on mobile platforms that can utilize the technology developed in this project.

The third party development creates slight difficulties. Depending on the technique the framework uses, it must have some way of identifying objects on the run. This might become an issue since most other than GUI based frameworks use web or native element identification to navigate through the application. As RAY's QA knows more or less as much of the software as any player, the solution ought to provide a methodology that introduces the application components. For example, it is impossible to click on a button if no information of the button is available and image recognition is not an option.

These criteria in mind different solutions were searched from the internet. At this point, literature reviews were not considered a good source for information as the industry is developing in fast pace and new technologies and solutions come out every now and then. Generally, there are two different technologies available that for the most parts fulfill the criteria mentioned above. Either the solution is GUI-based and elements are identified by images, or the solution is code-based and element identifiers are used. Naturally a combination of these solutions is possible. Based on these possibilities three, different options were drawn. These options are presented in the following three chapters.

3.2.2 Solution I – VNC servers and image based recognition

The first solution includes installing some Virtual Network Computing (VNC) software on the continuous integration machine running the tests and on all targeted mobile devices. VNC servers allow the screen of the device to be visible on the screen of another machine. The control machine can then forward mouse and keyboard actions to the target device. These server apps are also available for mobile devices either for free or for a nominal fee for all operating systems. The automated test cases are scripted in any selected framework that supports image and optical character recognition (OCR). The script looks up for the reflections of the mobile device's screen for certain images and performs all operations based on the images or text found in the images. One possible framework is to use the open source, Python-based Sikuli. Commercial options exists also, one of them being eggPlant Functional by TestPlant. The scripts can be executed on the framework itself, or the script execution can be outsourced to Robot Framework. If the framework does not support Jenkins integration for test run maintenance, the latter option is advisable. Implementing any other CI server than the already existing Jenkins would not be profitable, as long as the Robot Framework integration and thus Jenkins integration is easy to adapt.

Script reuse is a crucial part of test automation. It must be implemented to some extent in this solution. The problem, especially with mobile devices, is the screen resolution. E.g. the image of the login button on one device does not match the image of the button of a different resolution device. To clear this issue, the following solution was constructed.

In general, the test plan mostly contains only use case test scenarios. Therefore, actions performed on the device are similar to what a normal user would do: clicking, swiping, typing and element verification. These four actions are also the core actions that the automation scripts will perform in any framework. Therefore, only one function per each action is created. The function gets the element it should click as a parameter. To make the scripts reusable on all desired devices, picture libraries are created. Yet considering the log in button example, the login button library contains images of the login button each taken for different devices. In practice the functions get a library (or list) as a parameter containing all the images. The functions are scripted in a way that if

any of the images in the library received as a parameter is present on the screen, the actions are then performed on this precise image. This way, the exact same script can be extended to cover any other device as long as the image libraries contain the images of that device.

The action sequences are lumped together into higher level functions driven with parameters to cover a specific feature. In the end, a single test case can contain functions in multiple levels. Creating this kind of hierarchy is more complex and time consuming than to script rectilinear device-dependent cases, but after it has been scripted for the first time, it is effortless to reuse and much easier to maintain. As a downside of this method, not only the cases need to be maintained but also the image libraries are vulnerable to changes. Even minor changes in any visual appearances can result in problems in image recognition. As a second downside this method assumes that an image taken for one device does not match an image taken for another device having a completely different purpose. Having experienced the system under test, this scenario is unlikely to take place.

This solution is as close as test automation can get to real user actions. In the end, real users base their actions on images that they see on the screen. For the merit of this solution it has to be said that the required technology and know-how already exists in practice within RAY.

3.2.3 Solution II – Commercial all-in-one mobile automation tools

There are multiple tools dedicated for automated mobile testing. This solution would rely on harnessing a completely new framework for RAY's QA. These all-in-one tools provide everything needed for mobile test automation. In a way, eggPlant Functional by TestPlant presented in the previous section, is a tool of this type. Having experimented with multiple commercial tools, the most suitable tool came out to be SeeTest Automation by Experitest or Silk Mobile by Borland. In practice, these two are the same – the latter one is a rebrand of Experitest's product. In the following sections, the solution refers to SeeTest Automation though both programs are equally good.

SeeTest Automation comes with a more or less plug-and-play way of introducing things. The scripting language is highly intuitive and the UI is easy to use even for a non-technical tester. It includes views of the devices' screens, test recorders and execution tools and reporting features. On top of these, the greatest advantage still is the object spy feature. It allows the user to dismantle any GUI view to its elements and provides all native and web element information. This information is highly useful while scripting actions. The tool provides each element a unique reference path even in cases when an element is missing details. In this case, the path bases on the hierarchy of the elements.

Figure 7 demonstrates the object spy feature. The mobile casino view is reflected from an iPad and the object spy feature is activated. The blue squares represent elements that SeeTest Automation recognizes as web elements, yellow ones as native elements and the green one is the element currently selected, in this case the login button. On the right hand side, the hierarchical position of the login button is shown underneath the element details.

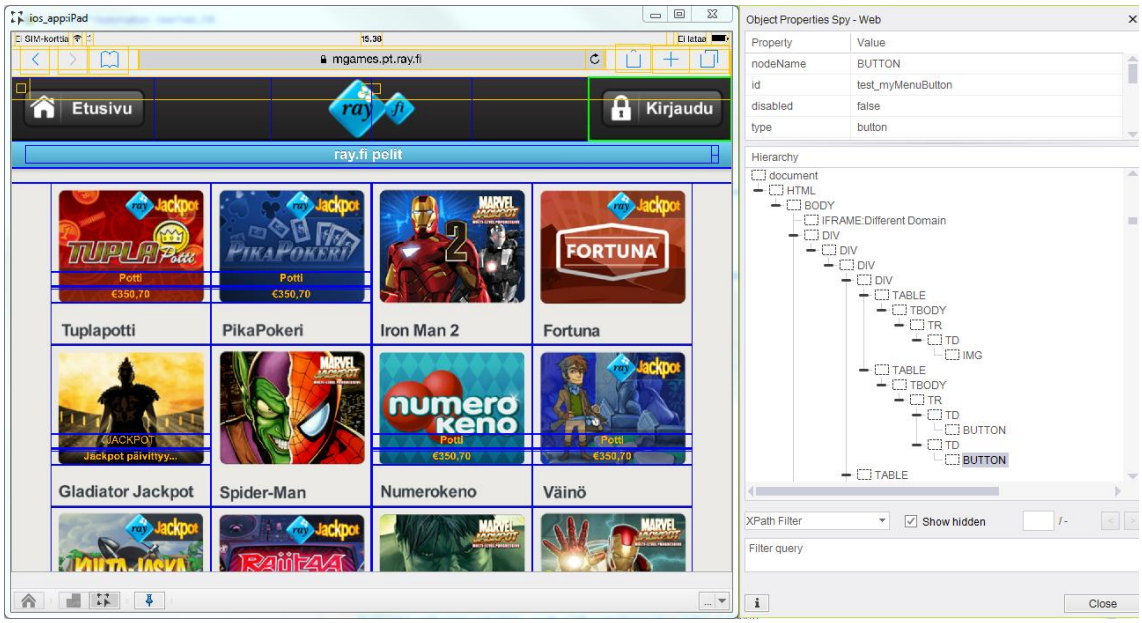

FIGURE 7. OBJECT SPY VIEW OF THE MOBILE CASINO'S LOBBY.

When scripting new test cases, the simplest way is to use the record function. It takes notes on all actions the user performs on the device and delivers them to a single script. In case the recorder refers to an element in a wrong way, the object spy can be used to find another way of referring to it. In practice, web elements have multiple properties that can be referred to: content, id, xpath (hierarchy based) etc. Depending on the situation, different references have unique advantages. In case an element is completely missing proper identification information, the software can perform image and text based recognition. Based on minor experimenting on the mobile application, this feature can prove to be useful in some special cases.

Although the product suits the project's needs in multiple ways, it is not without its own problems. First of all, it does not have support for any kind of keyword or function driven scripting which is an essential feature as noted in the Chapter 3.2.1. This shortcoming can be avoided as the software allows the test scripts to be extracted to various programming languages including Python. As mentioned earlier, RAY's QA already uses Python scripts in Robot Framework. Thus, these extracted scripts can be executed via Robot Framework. This allows also keyword driven scripting. In practice, this means that one test case from SeeTest Automation's perspective consists of multiple different tests i.e. one script contains only small parts of the real test case. A script could consist of only clicking on a single element or applying credentials to the login window. These small pieces are gathered into a single library file. The bits are then collected into Robot Framework into a single test case. The advantage is that this way the small pieces are reusable. If the goal is to have a parametrically driven script, the script is modified in a text editor to include parametrical properties. It can be said that the tests are all run on Robot Framework which uses SeeTest Automation as a driver. SeeTest Automation's UI is therefore only used for scripting purposes. It sounds complicated, but in the end it only requires the skill of copy-pasting text and creating scripts on SeeTest Automation. The latter requirement is easy to learn thanks to the user friendly GUI. Also, this methodology does not require integrating SeeTest Automation to Jenkis as the integration is done already on Robot Framework.

As a second downside, the software requires an annual license. The license fees depend on the machines in use and supported mobile operating systems. On the other hand, the

license includes online support, education material and all updates for the software. Still the biggest issue with SeeTest Automation is the support of different mobile browsers. It supports Google Chrome on Android and Safari on iOS which are both ideal browsers for testing as they are widely used. Unfortunately, on Windows Phone it does not support Internet Explorer as it supports only Simple Browser. It comes with limited features and does not support RAY's website (the focus of this project). Even if the browser supported the web site, it would be contradictory to test on a browser that is not officially supported and has very minimal distribution. In practice SeeTest Automation does not support Windows Phone in this case.

Relying on a software with an annually renewable license comes with certain risks. As the license is only annual, there is no guarantee what it will cost the next year. Most likely the fee will not skyrocket but the possibility is always present. In this case, the same software is provided by two suppliers which probably shows in the price competition. The situation is thus a little complicated as Borland is reliant of Experitest. The test scripts that can be extracted from the software are almost alike. The change from a supplier to another can be achieved by manipulating only a few lines of code. In the worst case scenario, if a change in suppliers were to take place, the process would require only the installation of new software and making the proper changes to those few lines of code.

3.2.4 Solution III – Combination of open source mobile automation tools

This solution incorporates the use of an open source mobile automation tool. There is a big variety of open source tools out there. The greatest advantage of open source tools is that they are usually free. There is a variety of platform-specific tools. Typically these tools are high tuned to give their all on that one specific platform and thus are efficient tools. Regarding the project at hand, these platform-dependent tools would most likely not be suitable. The use of any would either result in only one platform being automated or in the utilization of multiple frameworks. The first option is not desirable since in that case the automation would cover only minimal parts of testing leaving tremendous parts for manual testing. As learned in Chapter 2.5, the latter option would most likely result in a lot of maintenance work and thus contradict the reasons for automation. A combination of platform-dependent frameworks can be considered as an option in case of special characteristics.

In practice, the most suitable solution would most likely be to use one cross-platform tool. One major problem occurred though when searching through the internet for different frameworks: no cross-platform tool supports all required OS's for web application testing. Windows Phone is the source of all problems. Although some references of open source WP automation tools do exist, no proper cross-platform tools are available. Therefore, this solution focuses on tools that support solely iOS and Android. In the following sections, I will introduce the most suitable open-source tools for iOS and Android automation.

One method is to utilize one complete framework for automation testing, as in the second solution. MonkeyTalk is an open source tool that provides similar properties as SeeTest Automation. It includes an intuitive GUI, record and play options, data driven testing to mention a few. It suits native application automation well. It could be a viable option for this project otherwise, but is has one critical flaw. For web app testing, it supports only MTBrowser that simulates mobile Safari and Android native browsers. The lack of support of the Google Chrome and Safari browsers as well as the Android

native browser is a factor that cannot be dismissed. Thus, MonkeyTalk cannot be considered an option for this project.

A second option would be to harness a framework where all tests are hand coded. Suitable options could be Appium or Calabash. They both support real iOS and Android devices and all desired mobile browsers for web application testing. Because all tests need to be manually coded, they require a technical tester for developing and maintaining the tests. This is a complicated issue since in RAY's QA the tests could be developed by a non-technical tester. A suitable workaround for this issue is to create a Robot Framework library, which was suggested in the second solution (Chapter 3.2.3). Appium can be coded in a variety of languages, including Python and Java. These commands can then be executed on Robot Framework. Unfortunately, Appium's iOS automation cannot be done locally on a Windows machine since Appium relies on OS X-only libraries. Therefore, a Mac is needed at least for the host server. As Appium is widely used, proper documentation and community support is available.

As a rival for Appium, Calabash could be viable second option. It is coded in Cucumber – utilizing natural language. Thus, the language is easy to understand even for non-technical testers and business representatives. To ease maintenance efforts, Robot Framework integration would still be useful. Luckily, Android and iOS libraries are already available for Robot Framework. In practice, these libraries use Calabash Android and Calabash iOS Servers internally. So in a way, the integration is already done.

The Robot Framework mobile libraries have one specific downside: They are not that commonly used. Only very limited documentation and support is available for those libraries online. Although Robot Framework is widely used in RAY's QA, these specific libraries have never been used in RAY, and no internal knowledge exists.

A common problem with both Appium and Calabash is that they both utilize white box testing. In practice, this means that they require information about the elements, such as IDs or contexts, to function. In case of self-developed software, this is not an issue since you can control the elements' information. For a third party developed counterpart, it is not that simple. As neither Appium nor Calabash has built-in features for gathering this information, they need to be sought elsewhere. At least to some extent, the developer tools of the desktop version of Google Chrome can help. That way, element IDs or contexts can be fetched. For any extra information, GUI rippers can be used. As explained in Chapter 2.3.2, these tools dismantle the GUI into its elements. All needed element information can be retrieved from the ripper's results.

In this case, the extended use of GUI rippers would most likely not be a profitable choice. As learned earlier, rippers are efficient, creating hierarchical models of the GUI. Sometimes they might be even too efficient creating loads of useless data. In the test plan introduced in Chapter 3.1, the share of state transition tests between different pages is not that vast. Those tests are simple straightforward tests that are easy to automate manually. Incorporating automated GUI ripper tests would most likely result in additional maintenance work and poor profit. Therefore, it is suggested that GUI rippers are only to be used for gathering information, not for automating automated testing.

3.2.5 Weighing the solutions

Each of the presented solutions have their unique combination of pros and cons. Yet they share one common characteristic. All three solutions utilize some third party

developed software. Choosing any solution comes with the same threat; relying on a single framework, that is not in your control, includes risks in the areas of maintenance and support. If the third party decides to run down the support and development of the framework, also the use of the software is questionable. As long as the framework is not built completely internally within the corporation, this risk cannot be avoided. Luckily, this risk is fairly minimal in most of the cases since the solutions employ popular software. Nevertheless, the risk cannot be completely dismissed.

The effect of the plausible risk can, however, be minimized. Most solutions above utilize the use of keyword based Robot Framework. When the test cases are all scripted in Robot Framework which then drives another software to execute commands, it leaves the decision of the mobile framework open. Test case logic will always remain the same. This is due to the style tests are scripted in Robot Framework. One test case consists of executing sequenced keywords. If the keywords are scripted in a high level, they are completely independent of the mobile framework underneath. If, for some reason, the mobile framework ever needs to be revised, the only thing requiring calibration is the lowest level commands in Robot Framework. In a way, it is a delusion that this ideology would minimize the total risk of relying on a single software. The whole ideology is based on the assumption that Robot Framework would always be there for RAY QA to use. Despite this delusion, the ideology is presented as the transition to a Robot Framework free RAY QA would result in tremendous amounts of labor. In that event the problem of porting the mobile tests to a new environment would be the smallest of all problems.

By far, the simplest solution would be the second solution; to use commercial all-in-one test automation tools. The only thing that it cannot do, compared to the other solutions, is Windows Phone automation. Otherwise, there is nothing that the solution cannot do that another solution could. If Windows Phone were to be neglected from the equation and money were not an issue, it would make no sense to choose any other solution than the second one. The final decision of the solution rests on the following two questions: What is the status of Windows Phone now, and predicting the future, how will it stand after a couple of years? Is the second solution so much better than the open source solutions that it would be worth the licensing fees?

The only solution that would include automating also for Windows Phone is image based automation, i.e. the first solution. By far, the greatest advantage of image based automation is the quality information it can provide. Real life users' actions are solely based on the images they see on the screen. They do not know or even care what an element's ID is. Therefore, the best quality information would be provided by using image based automation. At least in a small scale it would be a good solution. Straightforward image based automation scripts are fast to develop. Image comparison even in a small scale unfortunately includes balancing between accuracy and efficiency. If the required comparison accuracy is set close to a 100 percent, there could be a great risk of missing existing elements. On the other hand, if the accuracy is set too low, there would be the opposite risk of detecting false elements. Managing both of the risks can sometimes be complicated.

In a large scale, image based automation has other issues. The scalability of the solution is only moderate. Of course, adding a new device to the testing laboratory does not mean scripting completely new tests, but it still requires the image library to be updated in case it is developed as presented in the first solution. In theory, plug and play is the only thing that is required in an element ID or context-based approach. A second issue in a large scale image based automation is its vulnerability to graphical changes.

Whereas element IDs rarely change, graphical aspects are more prone to change under development. In case a graphical element changes, it does not require capturing only one new image but the specific image needs to be captured on all devices under test.

When considering the future of Windows Phone support in RAY, things get complicated. As noted earlier, the gaming platform is provided by Playtech. They operate on international markets where iOS and Android are the only major mobile operating systems. Globally Windows Phone has only a tiny bit of the market. It is known though that in Finland Windows Phone is a huge success compared to the worldwide trend. However Windows Phone has not reached the selling numbers of its rivals even in Finland. Predicting Windows Phone's shares in the future is not within the scope of this work but Windows Phone can considered to come with certain risks. From Playtech's point of view, how long will they support an operating system with minimal distribution? Another thing to consider is the development activities so far. In RAY's mobile gaming site the variety of games for Windows Phone is considerably smaller than for iOS and Android.

Test automation is always a long-term investment. Development costs are big in the beginning, but the future savings on manual labor are so substantial that the investment will pay for itself in the long run. This, of course, is assuming that the automation is developed in properly. Because the automation solution that also supports Windows Phone includes certain disadvantages and the future of the platform is unclear, the conclusion is to leave out Windows Phone from the automation scope at this point. If, however, things change in the future, the question of automating Windows Phone should be revised, but for the moment it is not recommended. The decision was based on the uncertainty of the future of Windows Phone and hence the payback of its automation, and also the fear of instable automation tests if they were to rely solely on images.

That being said, the last major question that remains is whether to rely on commercial or open source tools. Are the commercial products worth their license fees? Looking from the outside, both solutions should look more or less the same in the end. Both solutions use element information and image comparison for testing. For the merit of SeeTest Automation, it bears mentioning that it can do also OCR. The decision was based on the following arguments:

- Ease of development
- Ease of maintenance
- The use of the framework outside this project.

Considering the development activities, the commercial products are clear winners. Although the tests will most likely be developed by technical testers, the commercial products offer a gateway for non-technical testers to develop the tests on the mobile platform. One challenge of the open source tools is the need for combining multiple tools to achieve a functioning solution. The record and Object Spy features of SeeTest Automation offer a simple way to develop tests as the program sees them, i.e. it is evident that the developer and the program share the same language. Combining different software that see things from different viewpoints can cause issues. As for the merit of the commercial products, also the following arguments can be stated:

- The setup is easier. Commercial products invest more in user-friendliness and therefore things work pretty much like plug and play.
- Development and maintenance activities are faster as the development happens within one single framework.

- Individual support is available all the time. When using free products, you can only hope that someone replies to your problem in an online forum.
- Commercial tools are easier to use. The reason lies in user friendliness.
- RAY plans to release a couple of mobile products on iOS and Android as Native applications. These products share the same characteristic with the mobile casino: they are developed by Playtech and therefore no white box information is available beforehand. These products could also be automated at least on some level. As these are native applications, the use of open source tools would require different GUI rippers than used within this project.

In a nutshell, it can be said that there are no areas where the open source tools beat the commercial ones, except for the fees. One downside of the commercial tools in this case is that you have to pay also for features you do not need. What the second solution suggests is that the automation would be executed via Robot Framework, and in the long run SeeTest Automation would be used only as a driver. Other functionalities would be used only for development purposes. Features like reporting tools, test execution tools and integration possibilities would never be used, even though they would be paid for.

Considering the scale of the project and comparing it to the licensing fees, the decision is clear: The licensing fees of the commercial tools will pay themselves back by better development quality and thus easier maintenance on top of better quality related information. Therefore, the use of commercial mobile automation tools, i.e. the second solution, is to be recommended.

As noted in Chapter 3.2.3, practically the same product is provided by two different suppliers: Experitest and Borland. There would not be any difference in choosing any of them, since the products are alike. Some factors that should, yet, be considered are

- pricing models
- support and other maintenance promises
- the scales tip slightly for Experitest's behalf since using Borland as a supplier would add an extra intermediary in the equation which can be seen as a minor risk.

After receiving individual offers from both supplier and balancing between the criteria mentioned above the decision was made: Experitest's SeeTest Automation combined with Robot Framework is the solution recommended in this thesis.

3.3 Proof of concept

3.3.1 Scripting test cases

So far, this thesis has introduced the mobile test plan including the test cases to be automated and the automation framework solution (Chapters 3.1 and 3.2). It was shown that a great deal of the cases of the test plan can be automated. When the test cases were scripted as was shown in Chapter 3.1, the number of test cases per operating system was 57. Number of test cases with automation prioritization A1 or A2 was 28. As Windows Phone was dropped out of the equation, the number of test cases to be automated in the first phase was $2 * 28 = 56$.

The 28 cases per OS consist of the 28 functionalities that are to be automated. In practice, the number of automation test cases is much higher. This is due to the convention that it is wiser to dismantle big test cases into multiple cases if they are

automated. This way, the results of the automation run are clearer and the tests themselves are more stable. A good example of dismantling tests for automation is case number six, *State transitions – Games – Open in real mode*, in the detailed test plan provided in Appendix 1. Manually performed, it is easier to just go through all the games and report them afterwards. When considering the automation of a test case, it is preferable to have a separate test case for each game. To reduce maintenance efforts, it is obviously preferable to create only one data-driven script which is capable of opening all games on each platform. As the two platforms – NGM and MGP – differ considerably they are to be separated as their own scripts. Although only two scripts exists for each, every game is opened separately i.e. all normal setups such as opening the page and logging in are done in between opening games. Applying this methodology to all tests, the number of automatable test cases rises tremendously. In the end, it should be kept in mind that the application of keyword and data-driven automation reduces maintenance and development efforts significantly.

The development guideline of new mobile automation test cases is described in Figure 9. This methodology represents the traditional way of developing automated tests. Later on in this chapter, a Test Driven Development model (TDD) will also be introduced. When a test case is automated, the needed scripts are created in SeeTest Automation in case they not already exist. After the scripts have been tested against the target device in SeeTest, the respective Python scripts are extracted to a single library file where all scripts reside. From SeeTest's point of view, each Python script is a separate function within a class. Therefore, the library file consists of functions beneath one class.

There are multiple ways of creating scripts in SeeTest Automation. The most straightforward way is to use the record function. When the record is in use, all actions performed on the mobile device are scripted to SeeTest step by step. It is by far the fastest way of scripting tests, but on the down side, some accuracy issues may occur. SeeTest guesses to which property of the element the user wants to refer. In simple situations, the guess is likely to be good enough, but the more complex the situation, the weaker the guess is. In most cases, a great deal of manual adjusting needs to be performed.

If all element information is known by the developer beforehand, the script can be manually coded using SeeTest's scripting language in the GUI. The scripting language is highly intuitive, and the user is guided through the scripting process step by step which makes manual scripting fairly simple. Figure 8 illustrates the scripting view of SeeTest Automation. The hardcore version of manual scripting is to bypass SeeTest Automation completely. If the developer is familiar with the calls on the code level, in this case Python, and required element details are known, the scripts can be written directly to the library using a text editor. This method can only be used in cases where the developer is sure what he or she is doing since any typo made into the code results in failure of all functions in the library.

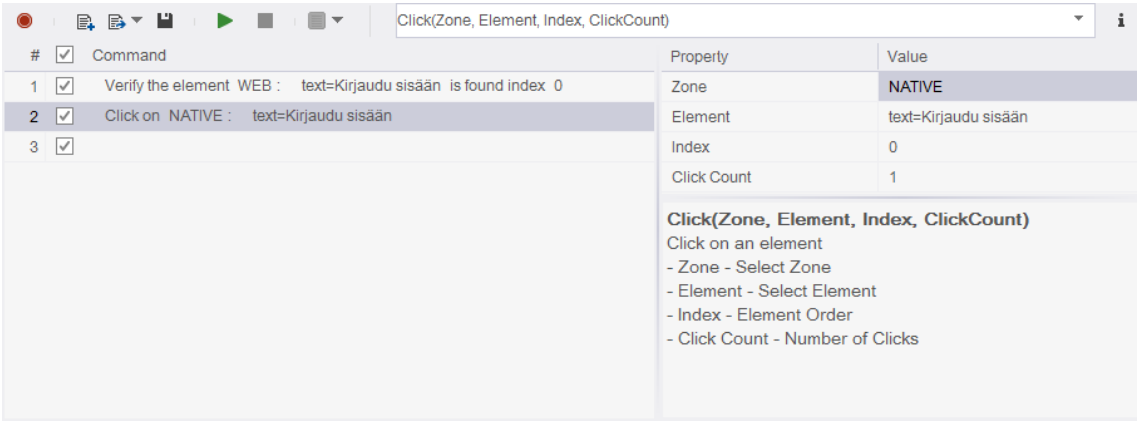

FIGURE 8. SEETEST AUTOMATION'S SCRIPTING VIEW.

The last option to develop scripts in SeeTest Automation is to use the Object Spy feature. It truly is useful, if no element details are known beforehand. As noted earlier, the Object Spy dismantles any GUI view on the mobile device to its elements. Browsing through the elements reveals all element details which can be used as reference. Going through GUI views one by one with the Object Spy and adding commands to desired element references is the best way of scripting stabile tests. It might be a bit slower than the other options, but at least all references and commands are as demanded.

On its own, SeeTest Automation is not an automation framework. It is more like a tool for developing tests, and a driver for executing the tests. It does not support data driven testing nor any kind of loops (e.g. if, while or for loops). Thus, the Python functions need to be modified later on in order to have parametrical properties to enable data driven testing if desired. This is achieved by editing the library file with any kind of notepad. It is the automation developer who is to decide, whether to create the code loops or if-statements to the library file, or to construct the cases so that these statements take place on Robot Framework. The optimal solution is highly case-dependent. Therefore, no general guidelines can be given.

RIDE, Robot Framework Development Environment, is used to create actual tests. In RIDE, the Python functions (and functions from other RF libraries as well) are combined to create keywords. These low-level keywords are to be kept quite short to maximize their reuse. Longer, and more case-dependent keywords can be created by combining multiple lower level keywords. When all required keywords have been created and properly documented, they are arranged in correct sequence to create a test case. In the last step, the test case is expanded to cover all desired devices. The test cases, keywords and Python functions should be developed in a way that they allow data driven testing, i.e. the same scripts can be reused on any device.

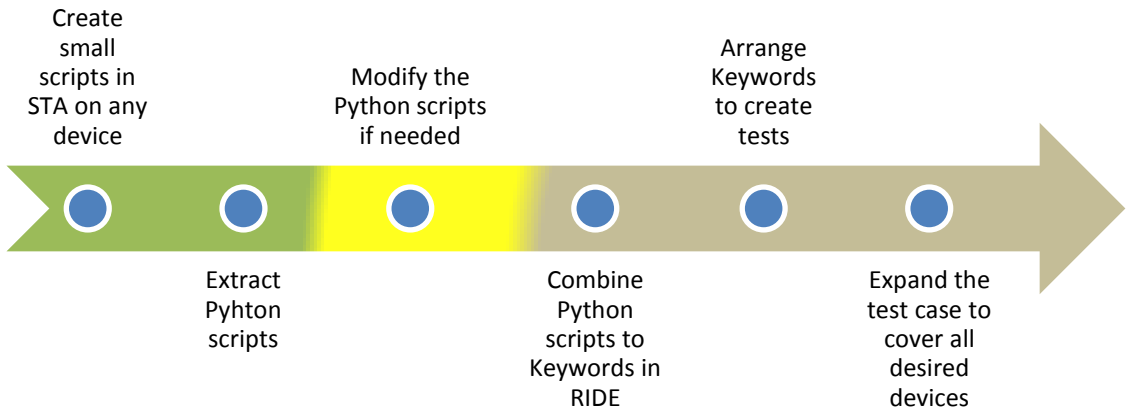

FIGURE 9. TRADITIONAL TEST CASE SCRIPTING TIMELINE. THE GREEN ACTIVITIES TAKE PLACE IN SEETEST AUTOMATION, THE YELLOW ONE IN TEXT EDITOR AND THE GREY ONES IN ROBOT FRAMEWORK.

A rival methodology of developing automated test cases with a test driven approach is presented in the following. The core of the methodology is based on conversations with *Juuso Issakainen, a co-tester* at RAY. These conversations aimed for delivering a highly documented way of creating test automation. In this thesis, the methodology is expanded to cover mobile test automation. It is drastically different than the traditional method. The arrow in Figure 9 is almost completely flipped around. The TDD approach is illustrated in Figure 10. The key difference between the methodologies is the way of building the test cases. In the traditional method, the test case building process starts from small pieces which are later on sewed together. The TDD approach does the exact opposite: it begins from the big picture and the low level steps are the last things to be specified.

In the Test Driven approach, the test case scripting starts from documenting the case deliverables. The documentation should answer to the following questions: Why is this case developed? What does it test? What major steps are included in the test? In which cases the test fails? In which cases does the test pass?

After proper documentation, the next step is to create high-level keywords in Robot Framework. These keywords can be completely imaginary, i.e. they do not need to be existing keywords. These imaginary or new keywords are documented properly in the same way that the test itself was documented: What does this keyword do? Why is this keyword developed? When does this keyword fail? When does this keyword pass?

When all keywords of the test case are documented and arranged, the keywords are given actual content. The content can either be other existing keywords or Python calls. If new Python calls are needed, they are scripted in SeeTest Automation. The scripts generated are then exported to the library file and modified in the same way they are treated in the traditional model.

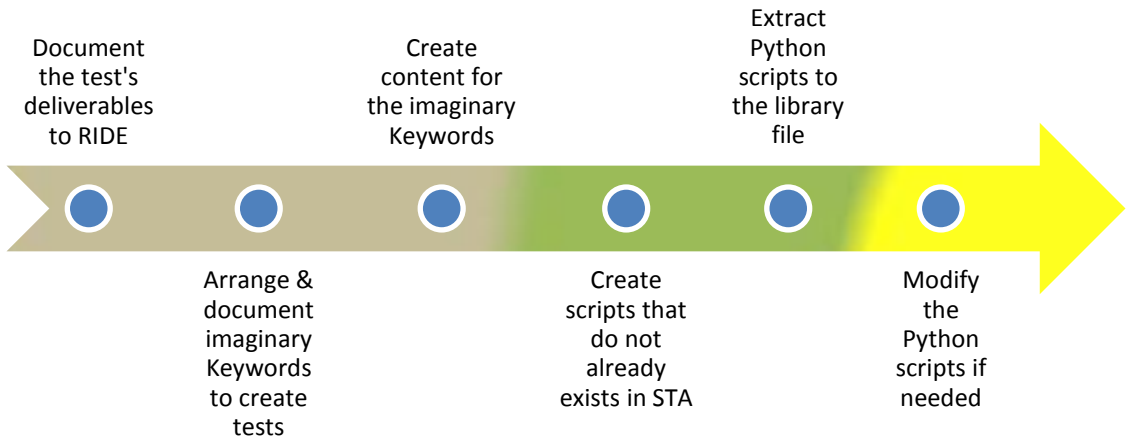

FIGURE 10. TEST DRIVEN DEVELOPMENT TIMELINE OF SCRIPTING NEW TEST CASES. THE GREEN ACTIVITIES TAKE PLACE IN SEETEST AUTOMATION, THE YELLOW ONE IN TEXT EDITOR AND THE GREY ONES IN ROBOT FRAMEWORK.

The TDD methodology is introduced in order to ensure the effective use of test automation. In the Chapter 2.3.1, Cem Carner's ideology was dicussed; harnessing test automation without a proper plan results easily in the trap that the automated features are those that are easiest to automate, not the ones that are most efficient in finding defects. The TDD approach presented is a variation of this ideology. It ensures that the developer is always clear on what is being done and why.

3.3.2 Executing test cases

Regardless of the applied ideology of developing test automation, the test cases need to face action. When a test case has been properly developed and successfully executed on all desired devices, it is equipped with different tags in demand. Tags are used to control the execution of test cases. In RAY's QA department, Jenkins is used to control the automation test execution and reporting. When a test case is signed with tags, Jenkins has the control over these cases. A test case can have multiple tags, for example *smoke*, *minor_regression* and *major_regression*. Jenkins can be triggered to run test cases with different tags. If, for example, a brief smoke test is to be run on all systems, Jenkins is triggered to run all test cases with the tag *smoke*. Triggering Jenkins can be manual or automated. In practice, the automatic triggering can be either periodical or Jenkins can search for component updates and run required tests when a component is updated.

Figure 10 illustrates the command flow taking place in RAY's QA systems, when automated mobile tests are run. When an automated test run is triggered in Jenkins, it commands Robot Framework on a slave machine to execute correct tests. Jenkin's slave machines are normal desktop computers that Jenkins can control. In this case, the slave machine is the same as the development machine. RF executes the tests. When an action is to be performed on the mobile device, RF executes Python scripts and then uses SeeTest Automation as a driver which runs in the background. STA then returns feedback from the executed steps to RF. RF can also run commands from its internal libraries. The co-work between RF and STA is looped until all tests are executed. After all tests are done, Jenkins searches for the test run results from RF and compiles a report. If all tests have passed, no further actions are required. However, if tests fail, the reason behind the failure is to be investigated.

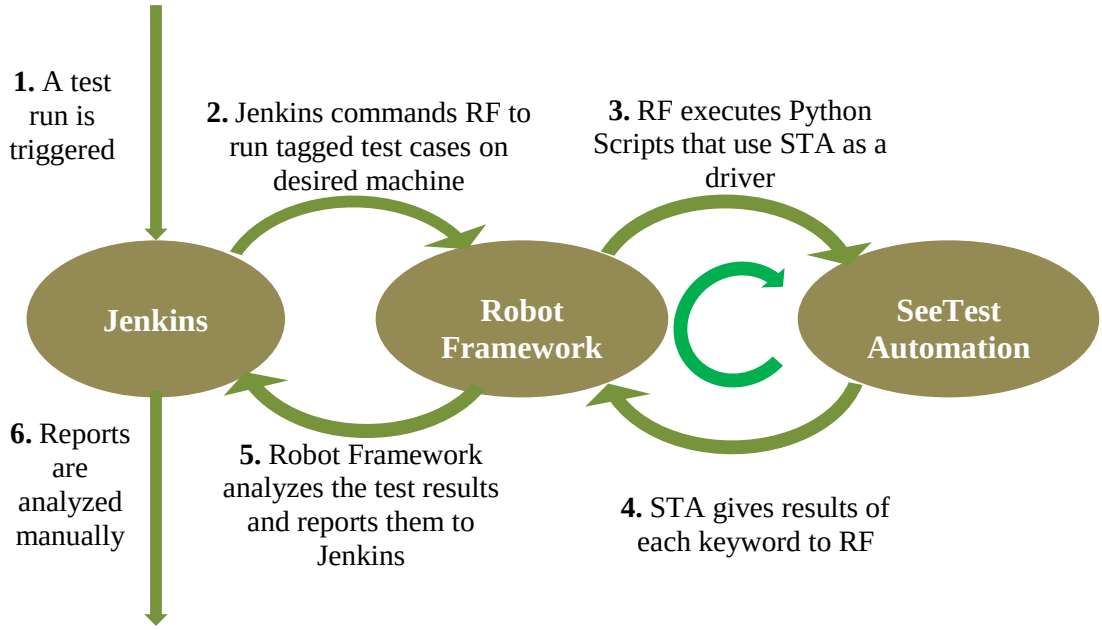

FIGURE 11. FLOW CHART OF AN AUTOMATED MOBILE TEST RUN. STEPS 3-4 ARE EXECUTED UNTIL ALL TESTS ARE COMPLETED.

It was earlier proposed that the automation should focus on iOS and Android. As it was shown in Chapter 2.4, the diversity of different device-OS combinations is tremendous and it is impossible to test the software on all of them. To optimize the device coverage and not to increase the amount of excess work, there should be some alteration between devices when the tests are run. One way of achieving this goal is to dedicate some devices for automation, and to perform manual tests on completely different devices. One SeeTest Automation's license can carry up to five different devices simultaneously. Thus, it might be advisable to reserve at least four devices, two of both OSs, for automation purposes. To optimize the usage, the devices should be of different resolution and have different versions of both operating systems. In practice, the most suitable solution is to choose one phone and one tablet of both OS, and to ensure that the OS versions differ.

As manual tests are performed on different devices than automated tests, the device-OS coverage should be rather good. Performing manual tests on 2-3 different devices per OS, the total coverage is 4-5 devices per OS. This amount is great enough considering that the whole test plan focuses mainly on functionalities. It is probable that other aspects are tested by Playtech on different diversity of devices and OSs.

3.3.3 The test scripts

The extent of the automation run is vast and therefore requires lots of manual labor. Within the limits of this project, only a proof of concept was conducted to show that the chosen automation framework fits the purpose. The POC includes the setup of the mobile lab and creating a small set of test cases to cover a few devices under automation. The test cases to be scripted within the POC are selected in so that they differ by technical execution as much as possible i.e. similar tests are not selected. Six test cases were selected for the POC:

- *1. Information Security – Login – Real*
- *6. State transitions – Games – Open in real mode* (scripted to cover only a few games)
- *7. State transitions – Lobby – Log out*

- *8. Gameplay – Games – Balance*
- *9. Responsible gaming – Gaming limits – Daily loss limit unexceedable*
- *17. State transitions – Games – Activate panic button*

The last three cases are the true scope of the POC. The first three cases contain functionalities that are indispensable for the purpose of automating the last three, and therefore they can be scripted also as separate test cases.

Appendix 2 lists the Python Mobile library which is the main library used in the RF scripts. The library has been developed using SeeTest Automation's all scripting methods listed earlier, except for the record function. In this particular case, the record function ended up being too unstable. Due to the amount of required manual adjustment, the advantage of recording was lost.

The Robot Framework scripts created for the POC are listed in Appendix 3. It contains all the tests listed above with one exception: The login and logout test cases were merged as one case, since merging the logout case to the login case extended the script length only by one line. The reason for a such small extension is that in order to log out of the service, one needs to first be logged in. It can be seen that the login case requires the logout case as a teardown and the logout case requires the login case as a setup.

In addition to the Mobile library, the RF suite uses only a few internal library calls and a couple of backend calls that already existed in RAY QA's libraries. These backend calls are API calls that are made to the Casino backend to retrieve player-related information. Using the API calls allows the test cases to contain less hard-coded information as most information required can be retrieved dynamically on the run.

Considering all the tests that are to be automated, a bunch of users with different profiles are needed. For example, the daily loss limit case (9^{th} case) requires a player with a minimal daily loss limit, whereas the gameplay test case (8^{th} case) requires gaming limits that are set high enough so that they are not unintentionally exceeded. Also, multiple tests operate with self-exclusions. Any activated exclusion prohibits the user from all gaming activities and therefore those accounts cannot be used for gameplay.

The best solution would be to create new users on the run with desired specifications, but unfortunately RAY's staging environment does not allow such events. Thus, a user profile library is created as a local resource file. Any use of the user profile library can be seen in the RF scripts in the following form:

${USERS['<userprofile>']['<information_needed>']}.

In practice, the user profiles are listed in one Python variable which is a library that consists of libraries. The outermost library consists of different user profiles. Every user profile is a library that contains the following information:

- **Username** used to log in to the service
- **Password** used to log in to the service
- **Password hash** (player's password in coded form) which is used in the API calls
- Player's **First name**
- Player's **Last name**.

Any other information needed can be retrieved on the run using the API-calls. Using the library, a unique user can be selected for each test's purposes.

To provide an example, case 17 *State transitions – Games – Activate panic button* is detailed in Appendix 4. In the appendix, every keyword or function is documented in a way that even those not familiar with RAY's mobile casino (but even once seen it) can follow up the logic how automated test cases are designed. It is noteworthy that this level of documentation is only done for demonstration purposes. The style of documentation provided in the Python library and the RF keywords is considered to be enough for RAY's internal documentation as it is assumed that test automation developers are even somewhat familiar with the product.

It was verified that all of the Robot Framework scripts were executed successfully with the devices intended in the proof of concept, in this case Google's Nexus 7 (Android 4.4.4) and Apple's iPad 3 (iOS 8.1.1). All scripts were also run with false parameters, or critical steps were removed from the tests to verify that tests fail if problems occur in the execution.

When creating the test cases for the proof of concept, two issues of SeeTest Automation came out. Firstly, while executing test cases on iOS machines, it turned out that the execution of the tests takes almost twice the time than on Android. Whereas one test on Android took from one to two minutes, on iOS the same test took from three to four minutes. Having a conversation with Experitest's experts, they informed that it is a feature of their software and nothing can be done about it. The second issue was that SeeTest Automation does not identify Android's native banner (the one including the back, home and app buttons in some devices) in the bottom of the screen. It became problematic in cases where the element that was to be clicked was under the banner. Instead of clicking on the element, the click fell on the banner. This issue had to be kept in mind while developing the test scripts.

Due to the two issues reported on SeeTest Automation, a proper way of developing automated test cases was delivered: For development purposes, an Android device without native hardware buttons (such as Nexus 7) would have to be used. Both issues point out this solution. Firstly, as the tests execute faster on Android than iOS devices, it is wiser to use Android for development since on the development phase any extra time taken for execution is a waste of time. Secondly, as Android has slightly more limits regarding the execution than iOS, Android's use on development is highlighted even more. When a test case is ready, it should naturally also run on the other devices in the lab.

4 Analysis and discussion

The intent of this thesis was to create a viable mobile test automation plan for RAY's QA department. It included a test plan for the mobile casino, selection of proper mobile test automation framework and conducting a proof of concept on it. All of these goals were accomplished.

The key factor in making any decisions on the test automation framework was Kumar Pradeep's ideology presented in Chapter 2.5: one needs to first clarify what is to be automated before choosing any framework. This is why a test plan needed to be created prior to making any decision on the test automation framework.

In the Aim of the Work, Chapter 1.1, there was no reference to the developer of the system under test. Yet, it was the aspect that affected decisions in all phases the most. Creating a test plan for a third party software is drastically different from creating one for an in-house software – not to mention developing test automation.

The differences will be discusse in the following section.

4.1 Test plan retrospect

The biggest difference between testing in-house and third party software is the role of testing in the development process. In well-organized in-house development projects, testing has a significant role throughout the process. It is the project manager's task to decide, whether the process relies on agile methodologies or whether the project is to be driven by some iterative waterfall model. Regardless of the development method, a testing specialist needs to be a part of the development team as early as possible. The earlier the needs of testing are recognized within the development team, the better.

Even in the most stubborn straightforward waterfall projects, testing needs to take part at the latest in the system testing phase of the process, but most likely already in the integration testing phase. Developers typically do their own unit tests, so testers do not need to contribute test execution activities at this phase, even though insight and guidance can always be beneficial for both parties. Therefore, the latest point when testing needs to take place in the process is when the integration tests need to be performed. Nevertheless, earlier would be better. Notwithstanding, testing is a multiphase activity in the process. There is much more to test than just to perform acceptance tests.

The nature of the different test phases stands out from one another. The focus point of every phase is different. Whereas acceptance testing focuses on the general behavior and user experience of the system under test, integration testing aims to find defects in the communication between components. So can acceptance testing be considered testing at all? Some references dictate testing to be an activity where defects of the software are pointed out. Hence, the answer to the question would be "no". In this thesis testing was defined as an activity of generating quality related information of the system under test. With this definition, the answer to the question is "yes".

In this particular case, testing had a complex role. When a new build is released, it is for the first time tested in an environment that even remotely resembles the production environment. For the first time, it faces RAY's interfaces and databases. The setup seems more like integration or system testing. Yet, the tests the build is about to face are

the last tests prior to release. Therefore, the definition of RAY's mobile casino testing discussed in section 2.6.3 holds up pretty well:

"Something between integration and regression testing with a severe user acceptance touch."

It is a completely different definition than in in-house projects. In this case, the testing team has only one shot in generating quality-related information. Also, the possibility of having discussions with the developers is practically zero.

The second great difference between testing in-house and on third party software is the "what" of testing. What is actually tested? Chapter 2.1.4 described a large variety of different things to test. In an in-house development project, all these aspects should be considered at least on some level. As it came out in the project at hand, the testing of functionalities was emphasized by far over all other aspects. It was the wish of RAY's business representatives. This standpoint can be considered fairly reasonable, even under closer scrutiny. There is no point in testing features that RAY has no control over. Some merits for the testing the third party performs need to be acknowledged. Although the decisions made were highly justified, it always leaves this question unsolved: does the test plan cover everything?

Especially in mobile testing, the answer to this rhetorical questions is rather simple. The test plan delivered surely does not cover everything. Even if the test coverage were perfect, imperfections appear at the latest in the device coverage. Each build is tested in the best case scenario with approximately a dozen device-OS combinations. Possible device-iOS combinations alone are counted in hundreds, not to mention Android devices where the number can be even millions. This is the clearest way of demonstrating the impossibility of testing everything. As a side note, it can be said that neither is the test coverage perfect. Hence, the rhetorical question should be formulated: does the test plan cover enough?

Putting it that way, the answer is more complex. Based on the existing knowledge within RAY, this plan has the best price-quality ratio. All business-critical components have been verified and even user-friendliness has been remotely taken into account in the test plan. When the automation run will be fully developed, the time taken for manual labor decreases significantly. Still, the test run as a whole is fairly extensive. Naturally, it may occur that a severe bug crawls through testing up to production. If such an event is to take place, there is no need to demoralize by the failure. A better option is to investigate whether the reason behind the failure was due to a dereliction in testing or if the test plan was flawed in some parts. If it was the latter, then the test plan needs to be revised. People learn by making mistakes. Regardless of how much resources are spent on testing, the risk of bugs slipping into production is always present.

The test cases that were tagged for automation with the automation prioritization A1-A4 were selected based on knowledge of the system under test. The best mileage was gained as two categories, A1-A2 and A3-A4 were formed. One can confidently say that the automation of the first category, A1-A2, will certainly be profitable. Making strict guidelines on which cases are automated and which not, makes no sense, since the knowledge of the automation framework is minimal. Why not let the boundary cases A3-A4 be left for later evaluation? In this case the boundaries were left quite wide, i.e. there are many cases either with A3 or A4 automation priority. Why not leave them wide? More sophisticated decisions can be made after more knowledge has been gained.

Considering the lessons learned so far in this section, the test plan can be considered to possess the highest return of investment.

4.2 Twisting through the automation framework

The selected automation framework included both commercial and free tools. The solution used is by far not the simplest possible by far as it requires a bunch of different components. The advantages of the solution probably outnumber its weaknesses and the advantages will be greater than in any other feasible solution.

The third party development also played a major role when selecting the automation framework. The solution was required to come with a method for gathering element information. It was a necessity, since Playtech does not hand over the information even for their clients. When weighing the different solutions, the lesson by Sridharan, presented in Chapter 2.5, was kept in mind: It is more convenient to select a framework that the staff is familiar with than to select a completely new language.

The justification behind the chosen solution was opened in Chapters 3.2.3 and 3.3. The solution chosen incorporates existing technology within RAY's QA department in almost as high degree as possible. Only Robot Framework's existing Android and iOS libraries would have used more of the technology already in use. In the end, that solution would have required other software to generate element information.

The greatest asset of the chosen solution is the keyword-driven and structure-based framework, which has also been suggested by Neal and Palani (Chapter 2.5). Structure-based frameworks are easy to maintain and the scripts have a high rate of reusability. The keyword-driven approach and the easy development of the Python scripts in SeeTest Automation enable even non-technical testers to take part in test automation development. This was not a primary requirement of the automation framework but a warmly welcomed side effect per se.

In the proof of concept phase, the example test cases were developed trying to hang on to the principle of a high reuse of the test scripts. There are as many development ideologies as there are developers. Thus, it is the developer's decision how the test scripts are formulated as long as in-house rules of development are being followed. The example cases were developed so that each Python script created was the shortest snippet possible to be used in some other case. As the Python script was being created, it was questioned whether the script could be parametrized. This ideology resulted in a highly reusable Python library. The Python scripts were then linked together in Robot Framework. The downside of the ideology is that the library became rather broad. Although the library is already quite long and will extend over time, it is still much shorter than if each test case would have been a separate Python script as it was intended in SeeTest Automation. Naturally, in that case the reuse of scripts would be impossible.

The other extreme would have been to create a replica of every SeeTest command into Robot Framework. Using this ideology, the Python library would have been the shortest and the script reuse the highest possible. It would have emphasized SeeTest's position as a driver even more, since the Robot Framework scripts would have been highly parametrical. As a downside of this ideology, the developer would have to know exactly how the elements are referred to. Of course, the Object Spy could be used but it would still leave quite a lot of work for hard coding. In this case, the natural language keywords would reside in Robot Framework instead of the Python library.

Although none of the development ideologies are completely flawed, some do fit certain situations better than others. In the example scripts, influences of all of the three ideologies can be seen, though the middle way is represented by far the most. As was already said, a similar and as good a result as the existing one would have been achieved by using the last mentioned ideology. The only great difference is, whether the main scripts reside in the Python or Robot Framework library.

4.3 General comments

The entire project was quite extensive. It required broad knowledge not only of development processes and the role of quality assurance in them, but also wide data processing skills – creating test automation without generating a single line of code would be an interesting process. In this project, the third party developed software provided a completely different approach to QA than in any other past projects. It was the aspect that affected all phases of the project the most.

From a learning perspective, this project served well. Taking a deep dive into the depths of testing and the role of QA in development processes was an eye-opening experience. The role of QA in development processes can be considered complex. In many cases, the role is diminished and QA is considered a mandatory activity in the end of any development project. In reality, the failure of QA is a great risk in any project. Still, it should be contemplated that the QA specialists are not the only ones responsible for the quality of the product. Quality is something that is built-in by every project team member.

The greatest lesson learned in this project was that default assumptions can be biased. As the project brief was given and before any literal inquiry was made, the project was separated into three sections: select a proper automation framework, generate a test plan and perform a proof of concept on the automation framework. It did not take long until it became clear that the setting was decided under false assumptions. Now it is known that there is no point in selecting the automation framework first and creating the test plan afterwards. Using this setting, especially in mobile projects, possesses multiple presumptions and shortcomings:

- The level of virtualization is attached without knowledge what is to be tested – simulators might not give enough quality related information or real devices could be way too heavy solutions
- The requirements of the automation framework are set completely based on assumptions. What if it turns out that performance measures are needed and the framework does not support them?
- It might turn up in the test plan that automation cannot be utilized in the project at all

In this project, the most critical bullet was the first one. The first setting assumed that mobile test automation can only be done on real devices. Although the end result relied on real devices, in the end, it does not mean that the first setting would have been right. In fact, in this case, the level of virtualization can be considered oversized. In Chapter 2.4, the justifications for different levels of virtualization were dictated. As the test plan focuses mainly on functionalities, even browser add-ons could have been considered viable options. It was the wish of almost all RAY's representatives that test automation should be conducted on real devices. Naturally simulators, not to mention real devices, provide better quality-related information. The automation process becomes heavier, and therefore more expensive, if lower level of virtualization is used.

From a system tester’s perspective, there was no need to object real device automation. It is a rare case when the representatives demand to allocate much more resources for a project than the leading engineer asks for. I was more than happy to welcome the increased project budget.

5 References

Aho, P., Kanstén, T., Doganay, K., Enoiu, E.P., Jääskeläinen, A., Sarunas, P., Limanauskiene, V. and Eldh, S., 2014. *Sate-of-the-art document: Processes and Methods for Advanced Test Automation of Complex Software-Intensive Systems.* ITEA 2 10037. Information Technology for European Advancement 2.

Álvarez, R.F., 2012. Testing Experience: Testing on a Real Handset vs.Testing on a Simulator –the big battle. **19**, pp. 42.

Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S. and Memon, A.M., 2012. Using GUI Ripping for Automated Testing of Android Applications, *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* 2012, ACM, pp. 258-261.

Bacj, J., 2003. Exploratory testing explained. *Online: http://www.satisfice.com/articles/et-article.pdf*, .

Balasubramaniam, V., 2014. The European Software Tester: Automation - The path to testing efficency. **6**(4), pp. 26.

Blackburn, M., Busser, R. and Nauman, A., 2004. *Why Mode-Based Test Automation is Different and What You Should Know to Get Started.* Software Productivity Consortium.

Borjesson, E. and Feldt, R., 2012. Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry, *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, April 2012, pp. 350-359.

Buonamico, D., 2014. Testing Experience: Enterprise Mobile Testing for Success. (19), pp. 60.

Burnstein, I., cop. 2003. *Practical software testing : a process-oriented approach.* New York: Springer.

Garner, M., 2011. Presenting with an iPad. *Journal of electronic resources in medical libraries*, **8**(4), pp. 441.

Grahrai, A., 2014. The European Software Tester: Barriers to test automation. **6**(3), pp. 22.

Grebenyuk, V., 2013. Testing Experience: Choosing the Right Test Automation Tool. (23), pp. 16.

Hass, A.M.J., 2008. *Guide to advanced software testing.* Boston: Artech House.

Hyysalo, S., 2009. *Käyttäjä tuotekehityksessä ; tieto, tutkimus, menetelmät.* Helsinki: Taideteollinen korkeakoulu.

ISTQB, 2011. *International Software Testing Qualifications Board: Foundation Level Syllabus.* International Software Testing Qualifications Board.

Karner, C., Bach, J. and Pettichord, B., 2002. *Lessons learned in software testing : a context-driven approach.* New York: Wiley.

Karner, C., Falk, J. and Nguyen, H.Q., 1999. *Testing computer software.* 2. ed. edn. New York (NY): Wiley.

Kensinton, K., 1997. *Software localization: Notes on technology and culture.* Cambridge, Massachusetts: Massachusetts Institute of Technology.

Kahn, M.E., 2010. Different forms of software testing techniques for finding errors. *International Journal of Computer Science Issues, May,* **7**(3), pp. 11-16.

Knott, D., 2014. Testing Experience: How to Stress Test Your Mobile App. (27), pp. 4.

Kuehlmann, A., 2014. The European Software Tester: The future of testing is automated. **6**, pp. 26.

Kumar, P., 2012. *Selecting the Right Mobile Test Automation Strategy: Challenges and Principles.* Cognizant.

Kumar, P., 2011. *Smart Phone Testing - Are You Smart Enough?* Cognizant.

Memon, A., Banerjee, I. and Nagarajan, A., 2013. *GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing.* Maryland, USA: University of Maryland.

Ministry of the Interior, 2013. *Ministry of the Interior's statute of gaming rules for the Finland's Slot Machine Association.* Statute edn.

Narashima, R.K.B., 2013. Testing Experience: Key Principles in Selecting the Right Automation Tools for Mobile Application Testing. (23), pp. 18.

Neal, C., 2013. Testing Experience: .NET Test Automation with Robot Framework. (23), pp. 6.

O'leary, A., 2008. Testing experience: Doing Automation the Smart Way. (4), pp. 42.

Opensignal, 2014. *Android Fragtmentation 2014.* Open Signal.

Palani, N., 2014. *Mobile Software Testing.* Mumbai, India: Wordit Content Design & Editing Services Pvt Ltd.

Playtech, 2014-last update, Playtech mobile hub. Available: https://www.playtech.com/products/mobile [10/06, 2014].

Pokerisivut, 2009, 16.10.2009. PAF:in huijarin tuomio lieveni hovissa ehdolliseksi. *www.pokerisivut.com*.

Rand, S., 2014. The European Sofware Tester: Is automation testing the future or a fad. **6:3**, pp. 19.

RAY, 2015-last update, RAY downloadable Poker client. Available: https://www.ray.fi/download_poker_client.

RAY, 2014a-last update, RAY's mobile casino. Available: https://m.ray.fi [10/06, 2014].

RAY, 2014b. *RAY's responsibility report and annual report 2013.* Espoo: Finland's Slot Machine Association.

RAY, 2011. *Financial statement 2010.* Espoo: .

RAY, 2010. *RAY announces license agreement with Playtech.* Espoo: .

RAYACHOTI, K., 2012. Testing Experience: Mobile Test Strategy. (19), pp. 30.

RBCS, 2012. *Advanced Software Testing: Solving Test Puzzles with Policies, Strategies, Plans.* Bulverde, Texas, USA: Rex Black Consulting Services.

Schultz, C.P., BRYANT, R. and LANGDELL, T., 2005. *Game testing all in one.* Boston, Mass: Thomson/Course Technology.

Sridharan, M., 2014. Tesing experience: Guidelines for Choosing the Right Mobile Test Automation Tool. **27**(27), pp. 60.

Sriramulu, V., Ramasamy, V., Jagadeesan, V. and Padmanaban, B., 2014. Testing experience: Mobile Test Automation - Preparing the Right Mixture of Virtuality and Reality. (27), pp. 46.

STATISTA, 2015-last update, Market volume of online gaming worldwide . Available: http://www.statista.com/statistics/270728/market-volume-of-online-gaming-worldwide/.

Suarez, M. and Aho, P., 2014. Testiautomaatio 2014: Käyttötapauksia ja tutkimusta, *Automaattinen regressiotestaus ilman testitapauksia*, 12.11.2014 2014.

Utting, M. and Legerad, B., 2007. *Practical Mode-Based Testing: A tools approach.* San Francisco, USA: Morgan Kaufmann.

Vidas, T., 2011. Toward a general collection methodology for Android devices. *Digital investigation,* **8**, pp. S14.

Whittaker, J.A., 2009. *Exploratory software testing : tips, tricks, tours, and techniques to guide test design.* Upper Saddle River, NJ: Addison Wesley Professional.

William Hill, 2015-last update, William Hill downloadable Poker client. Available: http://poker.williamhill.com/fi/download/.

Zhifang, L., Bin, L. and Xiaopeng, G., 2010. Test Automation on Mobile Device, *Proceedings of the 5th Workshop on Automation of Software Test* 2010, ACM, pp. 1-7.

List of appendixes

Appendix 1. Detailed test plan

Summary: This test plan is intended for new builds of RAY's mobile casino. The full extent of the set of test cases should be executed prior to every release. It does not matter whether the cases are executed manually or automatically. As automated tests have been created, corresponding manual tests can be given up.

Test environment: **RAY Staging**

Test entry criteria:

- The test environment is available and ready
- The build under test is uploaded to the test environment
- Testers have the needed rights in the test environment and backing systems

Test exit criteria:

- All tests are performed
- A guaranteed show stopper defect is found and resources can be allocated to other projects. If no other project needs the resources that were suddenly released from the mobile platform testing, the test run can be continued to find other plausible defects.

Tests are to be executed on all three supported operating systems: iOS, Android and Windows Phone

Level of automation:

- In the first automation phase 28 test cases shall be automated on iOS and Android. Whether to automated test cases with automation priority A3-A4, will be evaluated later.
- In this test plan automating Windows Phone is not recommended

Test cases

1.	Title: ***Information Security – Login – Real account*** Prioritization**:** *P1* Autom. prioritization**:** *A1* Description: *The case tests that player is able to log in to the game client with real account.*
2.	***Information Security – Login – Random credentials*** *P1/A1* *The case tests that logging in is not possible with credentials generated completely randomly.*
3.	***Information Security – Login – Blank credentials*** *P1/A1* *The case tests that logging in is not possible without inserting any credentials.*
4.	***Information Security – Login – Brute force login*** *P1/A1*

	The case verifies that brute force log in with a known username and randomly generated passwords is not successful.
5.	***Information Security – Login – Frozen account*** *P1/A2* *The case verifies that frozen accounts are not able to login to the system.*
6.	***State transitions – Games – Open in real mode*** *P1/A2* *The case opens all games in real mode.*
7.	***State transitions – Lobby – Log out*** *P1/A2* *The case verifies that the lobby's logout function works.*
8.	***Gameplay – Games – Balance*** *P1/A2* *A round of any game is played and verified that the GUI's balance view and the server side balance reacts to the gaming events correctly. The case is to be executed in both NGM and MGP platforms.*
9.	***Responsible gaming – Gaming limits – Daily loss limit unexceedable*** *P1/A2* *A bet that is to exceed player's daily loss limit is placed in any game. The test verifies that such bets cannot be placed. To be tested on both NGM and MGP platforms.*
10.	***Responsible gaming – Gaming limits – Monthly loss limit unexceedable*** *P1/A2* *A bet that is to exceed player's monthly loss limit is placed in any game. The test verifies that such bets cannot be placed. To be tested on both NGM and MGP platforms.*
11.	***Responsible gaming – Gaming limits – Game category "slot" unexceedable*** *P1/A2* *A bet that is to exceed player's slot game group limit is placed in any game. The test verifies that such bets cannot be placed. To be tested on both NGM and MGP platforms.*
12.	***Responsible gaming – Gaming limits – Game category "table" unexceedable*** *P1/A2* *A bet that is to exceed player's table game group limit is placed in any game. The test verifies that such bets cannot be placed. To be tested only on MGP platform as no table games belonging to the table game category exist.*
13.	***Responsible gaming – Session timer – Automatic log out option quits gaming*** *P1/A2* *The case verifies that if player has activated session timer with automatic log out option the game client automatically logs out after the time set is exceeded. To be tested on both NGM and MGP platforms.*
14.	***Responsible gaming – Session timer – Prompt message option interrupts gaming*** *P1/ A3*

	The case verifies that if player has activated session timer with prompt message option the game client automatically logs out after the time set is exceeded. To be tested on both NGM and MGP platforms.
15.	***Responsible gaming – Self-exclusion – Casino self-exclusion prohibits gaming*** *P1/A2* *A round of any game is tried to play when casino self-exclusion is active. All real money gaming activities is to be prohibited when self-exclusion is active. To be tested on both NGM and MGP platforms.*
16.	***State transitions – Lobby – Activate panic button*** *P1/A2* *In the case panic button is activated from the lobby view. Via backend tools it is verified that the self-exclusion has been activated.*
17.	***State transitions – Games – Activate panic button*** *P1/A2* *In the case panic button is activated from the game view. Via backend tools it is verified that the self-exclusion has been activated. To be tested on both NGM and MGP platforms.*
18.	***State transitions – Games – Log out*** *P1/A2* *In the case the functionality of the game view log out button is verified. To be tested on both NGM and MGP platforms.*
19.	***Gameplay – History – Data*** *P1/A3* *In the case a round of any game is played. All game event details are noted and compared to backend and player history details. To be tested on both NGM and MGP platforms.*
20.	***Information security – Lobby – Games not reachable without logging in*** *P1/N/A* *In the case exploratory means are used in trying to reach any game without logging in.*
21.	***Responsible gaming – Gameplay – Speed up gameplay disabled*** *P1/ N/A* *Using exploratory means for finding ways of speeding up the gameplay.*
22.	***Responsible gaming – Gameplay – Autoplay disabled*** *P1/N/A* *Using exploratory means of finding ways of enabling game play without user interaction.*
23.	***Information Security – Login – Remember me tic box*** *P2/A2* *The case verifies that if remember me tic box is activated on login, player's credentials are remembered on the next login, and vice versa.*
24.	***State transitions – Lobby – Header & Footer links*** *P2 /A2* *The case verifies the functionality and content of the lobby view links. The case is rated as a high priority test case since the links contain responsible gaming related information.*

25.	***State transitions – Lobby – Cancel panic button*** *P2 /A2* *The case tests that if panic button is selected, but not activated, no self-exclusions are activated.*
26.	***State transitions – Games – Cancel panic button*** *P2 /A2* *The case tests that if panic button is selected but not activated no self-exclusions are activated. To be tested on both NGM and MGP platforms.*
27.	***State transitions – Lobby – Deposit*** *P2 /A2* *The case verifies the correct behavior of the deposit page. Also the content and redirection to banks' pages are verified.*
28.	***State transitions – Lobby – Withdraw*** *P2/A2* *The case verifies the content of the withdrawal page.*
29.	***State transitions – Lobby – Timeout*** *P2/A2* *The case verifies that after a certain time of inactivity, player is logged out.*
30.	***State transitions – Games – Timeout*** *P2/A2* *The case verifies that after a certain time of inactivity, player is logged out. To be tested on both NGM and MGP platforms.*
31.	***Gameplay – Games – Broken Game*** *P2/A3* *In case within an incomplete game round player's connection brakes or gameplay is otherwise interrupted, a broken game is created. When the player resumes the game client the following time, the game should continue where it was left. Backend tools and player history should have entries of the broken game.*
32.	***Responsible gaming – Session timer – Prompt message allows extension*** *P2/A3* *In case player has activated session timer with prompt message option, player is notified when the set session time is exceeded. This prompt message allows either the session to be extended or ended. Gaming can be continued if the session is extended.*
33.	***Responsible gaming – Self-exclusion – Poker self-exclusion does not affect gaming*** *P2/A3* *The case verifies that if player has activated self-exclusion to poker games, no mobile casino gaming is prohibited. To be tested on both NGM and MGP platforms.*
34.	***Responsible gaming – Gaming limits – Daily limit about to exceed*** *P2/A3* *When a bet that is to exceed player's daily loss limit is placed, the system notifies player of the limit violation. The case verifies the pop-ups and that the system allows smaller bets until the limit is completely full. To be tested on both NGM and MGP platforms.*

35.	***Responsible gaming – Gaming limits – Monthly limit about to exceed*** *P2/A3* *When a bet that is to exceed player's monthly loss limit is placed, the system notifies player of the limit violation. The case verifies the pop-ups and that the system allows smaller bets until the limit is completely full. To be tested on both NGM and MGP platforms.*
36.	***State transitions – Games – Deposit*** *P2/A3* *The case verifies the state transition between any game and deposit page and back. To be tested on both NGM and MGP platforms.*
37.	***State transitions – Games – Withdraw*** *P2/A3* *The case verifies the state transition between any game and withdrawal page, and back. To be tested on both NGM and MGP platforms.*
38.	***Responsible gaming – Self-exclusion – Activation during ongoing gameplay*** *P2/A4* *The case checks that if self-exclusion is activated during an ongoing game round (e.g. a bonus round), the system allows the player to complete the ongoing round, and prohibits all following real money gaming activities. To be tested on both NGM and MGP platforms.*
39.	***Responsible gaming – Gaming limits – Game category "slot" limit about to exceed*** *P2/A4* *When a bet that is to exceed player's slot category loss limit is placed, the system notifies the player of the limit violation. The case verifies the pop-ups and that the system allows smaller bets until the limit is completely full. To be tested on both NGM and MGP platforms.*
40.	***Responsible gaming – Gaming limits – Game category "table" limit about to exceed*** *P2/A4* *When a bet that is to exceed player's table game loss limit is placed, the system notifies player of the limit violation. The case verifies the pop-ups and that the system allows smaller bets until the limit is completely full. To be tested only on MGP games as no table games root on the NGM platform.*
41.	***Gameplay – Games – Bet levels*** *P2/A4* *The case verifies that the bet levels of any game are as configured in the backend tools and that no bet level exceeds the level defined by the authority. To be tested on both NGM and MGP platforms.*
42.	***Games – Gameplay – General functionality*** *P2/N/A* *The games' general functionality is tested by exploratory means. Main focus is set on audiovisual elements of the game client. The lobby view and both NGM and MGP gaming platforms need to be gone through.*
43.	***State transitions – Games – Open in Fun mode*** *P3/A2*

	The case verifies that all games open in Fun mode.
44.	***State transitions – Games – Mode change from Fun to Real*** *P3/A2* *The case verifies that the mode change from Fun to Real is possible directly from the game view.*
45.	***State transitions – Games – Return to lobby*** *P3/A2* *The case checks that player is able to navigate from any game back to the lobby view. To be tested on both NGM and MGP platforms.*
46.	***Performance – Gameplay performance*** *P3/A3* *The case verifies that even after playing any game in a fast pace for a long time, it does not create any slowdown of the system. To be tested on both NGM and MGP platforms.*
47.	***Gameplay – Simultaneous use – Multiple instances on same account*** *P3/A4* *In the test the same account is logged in to the game client on multiple devices from different IP addresses without logging out in between. Player must be able to have only one active session open.*
48.	***Responsible gaming – Gaming limits – Loss limit violation during game round*** *P3 A4* *In the case player starts a game round and during the game round places a bet within the game that is to exceed any gaming limit. It is to be checked that no extra bets can be placed even during an active game round if they exceed any gaming limit. Gameplay can be continued without placing the extra bet. To be tested on Black Jack game as it is the only game allowing betting after the initial bets.*
49.	***Information security – Registration – via mobile*** *P3/N/A* *The case tests that a new player is able to register to the system with a mobile device.*
50.	***State transitions – Games – Customer service*** *P4/A3* *Player must be able to navigate to the customer service's pages and back in the game client. To be tested on both NGM and MGP platforms.*
51.	***State transitions – Lobby – Customer service*** *P4/A3* *Player must be able to navigate to the customer service's pages and back in the lobby view. To be tested on both NGM and MGP platforms.*
52.	***Information security – Login – Cancel login*** *P4/A3* *The test verifies that player is able not to choose to log in even after inserting account details*
53.	***State transitions – Games – Info*** *P4/A3* *The case verifies that player is able to navigate from the game client to game info pages in any game. To be verified in both NGM and MGP platforms.*

54.	***Gameplay – Simultaneous use – Desktop clients*** *P4/A4* *The case verifies that player is not able to be logged in on any desktop and mobile client simultaneously.*
55.	***Interruption – Push notifications during gameplay*** *P4/N/A* *The case tests the effect of push notifications and incoming phone calls to active game play.*
56.	***Interruption – Manually visit other apps during gameplay*** *P4/N/A* *In the test the effect of visiting other apps to the game client during active game round is analyzed.*
57.	***State transitions – Lobby – Remember audio*** *P4/N/A* *The functionality of remember audio option is verified. If audio is disabled from the lobby view, all audio should be muted everywhere in the game client and vice versa.*

Appendix 2. Python Mobile library

For this thesis, some information was concealed due to information security reasons. The concealed information is always replaced with a XXX symbol. For this appendix, the only concealed information is the URL of the mobile staging environment. Although the tests are run in the staging environment, the same tests could be executed in RAY's production site https://m.ray.fi. The libraries work also just as well on the production site.

```
-------------------------------------------------------------------------------------------------------------
----------------------
# -*- coding: utf-8 -*-

import unittest, sys
sys.path.insert(0, "C:\\Program Files (x86)\\Experitest\\SeeTest\\clients\\Python\\")
from ExperitestClient import Client
from ExperitestClient import Configuration

class Mobile(object):
   def SetUpDevice(self,device,open_casino_stg='True'):
      '''
         A mandatory element prior to any mobile test case. Currently available devices
are: Nexus 7 and iPad.
         By default the casino client is opened. It can be set not to open by
open_casino_stg==False.
      '''
      self.host = "localhost"
      self.port = 8889
      self.client = Client()
      self.client.init(self.host, self.port, True)
      self.client.setProjectBaseDirectory("C:\\Robot\\libs\\SeeTest_DB")
      if device=="Nexus 7":
         self.client.setDevice("adb:Nexus 7")
         if open_casino_stg=='True':
            self.client.launch("chrome:XXX", True, False)
      elif device=="iPad":
         self.client.setDevice("ios_app:iPad")
         if open_casino_stg=='True':
            self.client.launch("safari:XXX", True, False)

   def LogOutLobby(self):
      if(self.client.waitForElement("WEB", "id=test_myMenuButton", 0, 10000)):
            pass
      self.client.click("WEB", "id=test_myMenuButton", 0, 1)
      if(self.client.waitForElement("WEB",     "xpath=//*[@id='test_icon_logout']",     0,
10000)):
            pass
      self.client.sleep(500)
      self.client.click("WEB", "xpath=//*[@id='test_icon_logout']", 0, 1)
```

```
        if(self.client.waitForElement("WEB", "id=test_myMenuButton", 0, 10000)):
            pass
        self.client.verifyElementFound("WEB", "id=test_myMenuButton", 0)


    def VerifyClient(self):
        '''
            Verifies that the mobile casino client opened OK. Looks up for the myMenubutton and text "ray.fi pelit"
        '''
        if(self.client.waitForElement("WEB", "xpath=//*[@text='ray.fi pelit']", 0, 10000)):
            self.client.verifyElementFound("WEB", "id=test_myMenuButton", 0)
            self.client.verifyElementFound("WEB", "xpath=//*[@text='ray.fi pelit']", 0)
            pass
        else:
            self.LogOutLobby()


    def InsertCredentials(self,username,password):
        '''
            Inserts username and password to username and password fields respectively. As precondition the log in window needs to be opened. This can be done with function ClickMyMenuLobby().
        '''
        self.client.elementSendText("WEB", "id=test_usernameField", 0, username)
        self.client.closeKeyboard()
        self.client.elementSendText("WEB", "id=test_passwordField", 0, password)
        self.client.closeKeyboard()
        self.client.click("WEB", "text=Kirjaudu sisään", 0, 1)

    def ClickMyMenuLobby(self):
        if(self.client.waitForElement("WEB", "id=test_myMenuButton", 0, 10000)):
            pass
        self.client.click("WEB", "id=test_myMenuButton", 0, 1)
        if(self.client.waitForElement("WEB", "id=test_usernameField", 0, 30000)):
            pass

    def VerifyTextFound(self,text_to_be_found):
        '''
            Verifies that a string text_to_be_found in some element in the GUI view. Fails if the element is not found.
        '''
        xpath = unicode("xpath=//*[@text=" + text_to_be_found + "]")
        #if(self.client.waitForElement("WEB", "xpath=//*[@text={:s}]".format(text_to_be_found), 0, 15000)):
        if(self.client.waitForElement("WEB", xpath, 0, 15000)):
            pass
        self.client.verifyElementFound("WEB", xpath, 0)

    def VerifyTextNotFound(self,text_not_to_be_found):
        '''
```

```
        Verifies that no element containing the string text_not_to_be_found is found on the screen. Fails if the string IS found.
        '''
        xpath = unicode("xpath=//*[@text=" + text_not_to_be_found + "]")
        self.client.verifyElementNotFound("WEB", xpath, 0)



    def OpenGame(self,game,mode='Real'):
        '''
            Opens any game in the mobile lobby based on its game name. Games can be opened in mode=='FUN'. Changes also the screen orientation right.
        '''
        if game == 'Jacks or Better':
            if(self.client.swipeWhileNotFound2("Down", 700, 200, "WEB", "xpath=//*[@id='test_po' and @width>0]", 0, 200, 5, True)):
                pass
        elif(self.client.swipeWhileNotFound2("Down", 800, 500, "TEXT", game, 0, 200, 5, True)):
            pass
        if mode=='FUN':
            text='Kokeile ilmaiseksi'
        else:
            text='Pelaa rahalla'
        if(self.client.waitForElement("WEB", "text={:s}".format(text), 0, 10000)):
            pass
        self.client.click("WEB", "text={:s}".format(text), 0, 1)
        # the two lists below describe the games that are played in portrait/landscape. The last piece of the function alters the device into correct orientation.
        landscape = ['Football Carnival','Väinö','Spider-Man','Iron Man 3','Gladiator Jackpot','Tuplapotti','Numerokeno','Rautaa Pokeri','PikaPokeri','Desert Treausre','Rautaa','Kulta-Jaska',"Captain's treasure",'The Incredible Hulk','Iron Man 2','Fantastic Four','Mr. Cashback']
        portrait = ['Fortuna',"Santa's Gifts", 'Fish-O-Rama','Vacation Station','Chinese Kitchen','Pajatso']
        if game in portrait:
            self.client.sendText("{portrait}")
        else:
            self.client.sendText("{landscape}")


    def VerifyGame(self,orientation=0):
        '''
            Verifies that the game has opened by searching for id = test_balanceText. Also swipes up to activate full screen mode if needed.
            If the default orientation is not sattisfying, the orientation can be changed by the optional argument orientation == (landscape,porttrait).
        '''
        if orientation!=0:
            self.client.sendText("{orientation}")
        if(self.client.waitForElement("WEB", "xpath=//*[@nodeName='CANVAS' and ./parent::*[@nodeName='TD']]", 0, 10000)):
```

```
            self.client.sleep(500)
            self.client.swipe2("down", 300, 500)
            pass
        if(self.client.waitForElement("WEB", "text=Please choose table limits forCLASSIC ROULETTE", 0, 1000)):
            self.client.click("WEB", "text=Pelaa", 0, 1)
            pass
        self.client.verifyElementFound("WEB", "id=test_balanceText", 0)


    def NavigateMgp(self,whereto):
        ""
            This keyword can be used to navigate in any MGP game's menu. Possible whereto arguments are:

            home, info, support, deposit, withdraw, panicbutton, audioON, audioOFF, logout
        '''
        # this section is separated due to the separated home and info buttons in some mgp games that are not under the menu
        mgpquickbuttonlist = ['home','info']
        status = 0
        complicated_mgp='False'
        if(self.client.waitForElement("WEB", "xpath=//*[@id='test_homeButton'and @onScreen='true']", 0, 2000)):
            complicated_mgp='True'
        if whereto in mgpquickbuttonlist and complicated_mgp=='True':
            self.client.sleep(2000)
            self.client.click("WEB", "xpath=//*[@id='test_{:s}Button']".format(whereto), 0, 1)
            status ='passed'
            pass

        # and here starts the part for the rest of the MGP games (and other buttons for games mentioned above)
        else:
            if complicated_mgp=='True':
                if(self.client.waitForElementToVanish("WEB", "xpath=//*[@id='test_notification']", 0, 7000)):
                    pass
                self.client.sleep(500)
                self.client.click("WEB", "xpath=//*[@id='test_myMenuButton']", 0, 1)
            else:
                self.client.dragCoordinates2(30, 150, 420, 150, 500)
            elements_too_low = ['panicbutton','audioON','audioOFF','logout']
            self.client.sleep(1000)
            if whereto in elements_too_low and complicated_mgp =='False':
                self.client.elementSwipe("WEB", "xpath=//*[@id='test_icon_home']", 0, "Up", 600, 1000)
                self.client.sleep(2500)
```

```
        if(self.client.waitForElement("WEB",
"xpath=//*[@id='test_icon_{:s}']".format(whereto), 0, 3000)):
            self.client.click("WEB",    "xpath=//*[@id='test_icon_{:s}']".format(whereto),
0, 1)
            pass

    def VerifyBalance(self,balance_text):
        '''
            Verifies that an element with id = test_balanceText/test_balanceValue and text =
<balance_text> is found. Please note that the value is submitted without the € symbol.
        '''
#       if(self.client.waitForElement("WEB", "id=test_balanceText", 0, 1000)):
#                                   xpath = unicode("xpath=//*[id=test_balanceText and
contains(@text,'{:s}')]".format(balance_text))
#       else:
#                                   xpath = unicode("xpath=//*[id=test_balanceValue and
contains(@text,'{:s}')]".format(balance_text))
        xpath                   =                   unicode("xpath=//*[contains(@id,'alance')and
contains(@text,'{:s}')]".format(balance_text))
        self.client.verifyElementFound("WEB", xpath, 0)


    def StartJbRound(self):
        '''
            A round of Jacks or Better with a €0,10 minimum bet is started. This funciton
can adjust the bet from any previous bet. The game needs to be opened prior this
function.
        '''
        self.client.click("WEB", "id=test_decrease", 0, 5)
        i=0
        while i<7:
            if(self.client.waitForElement("WEB", "Kokonaispanos: €0,10", 0, 500)):
                i=10
            else:
                self.client.click("WEB", "id=test_betOne", 0, 1)
                i+=1
        self.client.click("WEB", "id=test_deal", 0, 1)
        if(self.client.waitForElement("WEB", "id=test_holdButton0", 0, 10000)):
            pass
        self.client.verifyElementFound("WEB", "id=test_holdButton0", 0)

    def PlaceJbMaxBet(self):
        '''
            Starts a round of Jacks or Better with max bet. Please note that this function
does not complete the game as it only starts the round. The game needs to be opened
prior this function.
        '''
        self.client.click("WEB", "id=test_increase", 0, 5)
        self.client.click("WEB", "id=test_betMax", 0, 1)
        self.client.click("WEB", "id=test_deal", 0, 1)
```

```
    def CompleteJbRound(self):
        '''
            Completes an already started round of Jacks or Better by clicking the deal button. Any plausible win is collected.
        '''
        self.client.click("WEB", "id=test_deal", 0, 1)
        if(self.client.waitForElement("WEB", "id=test_collectButton", 0, 3000)):
            self.client.click("WEB", "id=test_collectButton", 0, 1)
            pass
        elif(self.client.waitForElement("WEB", "id=test_collectLabel", 0, 3000)):
            self.client.click("WEB", "id=test_collectLabel", 0, 1)
            pass
        self.client.verifyElementNotFound("WEB", "id=test_holdButton0", 0)

    def VerifyPanicbuttonPopup(self):
        '''
            Verifies that the panicbutton pop-up has right content.
        '''
        self.client.verifyElementFound("WEB", "text=Olet valinnut itse asetetun pelikiellon oikean rahan pelitoimintoihin seuraaviksi 12 tunniksi. Vahvista valitsemalla \"Hyväksy\" tai valitse \"Peruuta\", jos haluat jatkaa pelaamista.", 0)

    def ApprovePanicbutton(self):
        '''
            Approves self exclusion confirmation by clicking "Hyväksy", verifies that the approval pop-up is right and closes the approval pop-up from the x symbol
        '''
        self.client.click("WEB", "text=Hyväksy", 0, 1)
        self.client.verifyElementFound("WEB", "text=Olet sulkenut pääsysi rahapeleihin", 0)
        self.client.click("WEB", "id=test_btnClose", 0, 1)

    def VerifyDailyLossLimitAboutToExceedPopup(self,dailyremaininglosslimit):
        dailyremaininglosslimit = "{0:.2f}".format(float(dailyremaininglosslimit))
        self.client.verifyElementFound("WEB", u"text=Panostuksesi ylittäisi asettamasi päivittäisen pelirajan. Voit panostaa enintään {:s} EUR.".format(dailyremaininglosslimit), 0)
        self.client.click("WEB", "id=test_btnClose", 0, 1)
```

Appendix 3. Robot Framework test suite

This appendix consists of the Robot Framework test suite that was constructed for the proof of concept. It bears mentioning that the Mobile Python library is the only one that is opened in detail within this work. This is due to the fact that the other libraries are not constructed within the framework of this work. The primary external resource that this suite uses is the IPJ_sanat.txt. From that library, keywords such as *IPJ – get player info* and *IPJ- get rg info* are used. These are API-calls that are put in the casino backend to get player related data. The pieces of data are used in test cases to verify the software's correct behavior.

```
*** Settings ***
Library         AutoItLibrary
Library         OperatingSystem
Library         Selenium2Library
Library         SIKULI_Sanat.py
Library         String
Library         Screenshot
Library         RequestsLibrary
Library         TC_Sanat
Library         Collections
Library         ../../libs/Mobile.py
Library         XML
Resource        ../../resource_files/IPJ_sanat.txt

*** Test Cases ***
Login – With real account
   [Documentation]    This test case verifies that palyer is able to log in to the game client. When player has been successfully logged in player is logged out from the service.
   …
   …    The test case fails if:
   …    – game client does not open
   …    – player's status is wrong at some point i.e. OFFLINE when logged in or vice versa
   …    – player's whole name is not visible in the front page after the log in
   …    – some button is input field is missing
   …
   …    – TL –
   Set Up Device    Nexus 7
   Verify Client
   Log in and verify state    ${USERS['Personal']['Username']} ${USERS['Personal']['Password']}    ${USERS['Personal']['Passwordhash']} '${USERS['Personal']['Firstname']} ${USERS['Personal']['Lastname']}'
   Log out and verify state    ${USERS['Personal']['Username']} ${USERS['Personal']['Passwordhash']}
   Verify Text Found    'ray.fi pelit'
   Verify Text Not Found    '${USERS['Personal']['Firstname']} ${USERS['Personal']['Lastname']}'
```

```
Open MGP game in real mode
   Set Up Device    Nexus 7
   Verify Client
   @{gameslist}=    create list    Fish-O-Rama    Kulta-Jaska    Classic Blackjack
   ${listlength} =    get length    ${gameslist}
   : FOR    ${i}    IN RANGE    ${listlength}
   \                Log    in    Mobile                ${USERS['Personal']['Username']}
${USERS['Personal']['Password']}                '${USERS['Personal']['Firstname']}
${USERS['Personal']['Lastname']}'
   \    Open Game    @{gameslist}[${i}]
   \    Verify Game
   \              Verify    MGP    balance              ${USERS['Personal']['Username']}
${USERS['Personal']['Passwordhash']}
   \    Navigate Mgp    logout
   \    Verify Text Found    'ray.fi pelit'

MGP Gameplay
   [Documentation]    This test palys one round of Jacks or Better and verifies when the
round is started:
   …    – Balance is deducted both in IMS and game view
   …    – Daily loss limit is deducted
   …    – Monthly loss limit is deducted
   …    – Gamegroup 2imit is deducted
   …
   …    Information of features listed above are generated via Backend API prior and
after the game round has started. These pieces of data are compared in the end of the
test and verified that the latter ones are deducted by the game round's bet(==0,10€).
   Set Up Device    Nexus 7
   Verify Client
   Log    in    mobile                ${USERS['Personal']['Username']}
${USERS['Personal']['Password']}                '${USERS['Personal']['Firstname']}
${USERS['Personal']['Lastname']}'
   Open game    Jacks or Better
   Verify game
   @{rg_data_beginning}    =    IPJ    –    Generate    loss    limit    data
${USERS['Personal']['Username']}    ${USERS['Personal']['Passwordhash']}
   Start Jb Round
   @{rg_data_middle}    =    IPJ    –    Generate    loss    limit    data
${USERS['Personal']['Username']}    ${USERS['Personal']['Passwordhash']}
   Complete Jb Round
   Navigate MGP    logout
   Verify Text Found    'ray.fi pelit'
   Compare float lists    ${rg_data_beginning}    ${rg_data_middle}    0.1

Activate Panic Button – MGP Gameview
   [Documentation]    This test case activates panic button via NGM game view. It is
also verified that the:
   …    – self exclusion expiration time differs from the one in the beginning of the test
   …    – panic button pop-up is correct
   Set Up Device    Nexus 7
```

Verify Client
@{self_exclusion_end_date_beginning} IPJ – Generate self exclusion data ${USERS[‘Personal’][‘Username’]} ${USERS[‘Personal’][‘Passwordhash’]}
Log in Mobile ${USERS[‘Personal’][‘Username’]} ${USERS[‘Personal’][‘Password’]} ‘${USERS[‘Personal’][‘Firstname’]} ${USERS[‘Personal’][‘Lastname’]}’
Open game Kulta-Jaska
Verify game
Navigate Mgp panicbutton
Verify Panicbutton Popup
Approve Panicbutton
Navigate Mgp logout
Verify Text Found ‘ray.fi pelit’
@{self_exclusion_end_date_end} IPJ – Generate self exclusion data ${USERS[‘Personal’][‘Username’]} ${USERS[‘Personal’][‘Passwordhash’]}
run keyword if @{self_exclusion_end_date_beginning} == @{self_exclusion_end_date_end} FAIL Self exclusions were not updated by the activated panic button

Daily loss limit not exceedable – NGM
[Documentation] This test case verifies that player can not place a bet that is higher than his daily loss limit. A 10€ round of Jacks or Better is tried to be played.
…
… The test case verifies that the:
… – The game round is not started i.e. player’s balance is not deducted and no bet reservations are made.
… – The pop-up appearing has right content
… – Player is able to continue gaming with a smaller bet
…
… Preconditions for the case:
… – Player’s daily loss limit is lower than 10€
Set Up Device iPad
Verify Client
Log in Mobile ${USERS[‘Personal’][‘Username’]} ${USERS[‘Personal’][‘Password’]} ‘${USERS[‘Personal’][‘Firstname’]} ${USERS[‘Personal’][‘Lastname’]}’
Open game Jacks or Better
Verify game
${remaining_daily_loss_limit} = IPJ – Get Rg info ${USERS[‘Personal’][‘Username’]} ${USERS[‘Personal’][‘Passwordhash’]} dailyremaininglosslimit
Place Jb max bet
IPJ – Verify bet reservations ${USERS[‘Personal’][‘Username’]} ${USERS[‘Personal’][‘Passwordhash’]} 50
Verify daily loss limit about to exceed popup ${remaining_daily_loss_limit}
Play round of Jacks or Better ${USERS[‘Personal’][‘Username’]} ${USERS[‘Personal’][‘Passwordhash’]}
Navigate Mgp logout
Verify text found ‘ray.fi pelit’

*** Keywords ***

```
Log in and verify state
    [Arguments]    ${username}    ${password}    ${passwordhash}    ${name}
    [Documentation]    Preconditions: Mobilecasino is in lobby view
    …
    …    Log ins to Mobile Casino and verifies the player's status prior and after the log in. Fails if :
    …    – My menu or log in buttons are not found
    …    – Player's status is ONLINE prior to log in
    …    – Player's status is OFFLINE after log in
    …    – Player's name is not found on the front page
    …
    …    Player's name is to be inserted in the form of:
    …    <First name><white space><Last name>
    ${status_beginning} =    IPJ – Get player info    ${username}    ${passwordhash}    status
    Run keyword if    '${status_beginning}' == 'online'    FAIL    'User was already logged in'
    Click My Menu Lobby
    Insert Credentials    ${username}    ${password}
    log    ${name}
    Verify Text Found    ${name}
    ${status_end} =    IPJ – Get player info    ${username}    ${passwordhash}    status
    Run keyword if    '${status_end}' == 'offline'    FAIL    'Login was unsuccessful'

Log out and verify state
    [Arguments]    ${username}    ${passwordhash}
    Log Out Lobby
    ${status} =    IPJ – Get player info    ${username}    ${passwordhash}    status
    Run keyword if    '${status}' == 'online'    FAIL    'Player is still online'

Log in Mobile
    [Arguments]    ${username}    ${password}    ${fullname}
    [Documentation]    Preconditions: Casino lobby needs to be opened and visible.
    …
    …    Logs in to mobile casino and verifies that player's full name is found on the main screen after login.
    Click My Menu Lobby
    Insert Credentials    ${username}    ${password}
    Verify Text Found    ${fullname}

Verify MGP balance
    [Arguments]    ${username}    ${passwordhash}
    [Documentation]    Retrieves player's balance from PT backend and verifies that in the mobile device's view an element with id=test_balanceText and text=<player's balance> is present.
    ${balance} =    IPJ – Get player info    ${username}    ${passwordhash}    balance
    ${pre}    ${post} =    Split String    ${balance}    .    1
    set test variable    ${balance formatted}    ${pre},${post}
    Verify Balance    ${balance formatted}

Compare float lists
```

[Arguments] ${first_list} ${second_list} ${differential} # The argument ${differential} is the difference parameter that the list items should have.
[Documentation] This keyword compares to lists. The list's items should differer by the differential parameter respectively. If the difference of any two item's is something else than the differential parameter the keyword fails.
${list_length} get length ${first_list}
: FOR ${element} IN RANGE ${list_length}
\ ${bet_deducted_beginning} = Evaluate @{rg_data_beginning}[${element}] - ${differential}
\ ${status} = Evaluate @{rg_data_middle}[${element}] == ${bet_deducted_beginning}
\ run key word if ${status} == False FAIL The difference between lists' two items was not equal to the diffential parameter

Play round of Jacks or Better
[Arguments] ${username} ${passwordhash}
[Documentation] Plays one round of Jacks or Better with 10c bet.
…
… After the round has started it is verified that a 10c betreservation has been made for the user. When the round ought to be completed a verification is made that there are no open betreservations i.e. player's currentbet == 0.
…
… Preconditions:
… – The game is verified to be open. Keywords Open Game (Jacks or Better) and Verify Game are advised to be used.
Start Jb round
IPJ – Verify bet reservations ${username} ${passwordhash} 50.10
Complete Jb round
IPJ – Verify bet reservations ${username} ${passwordhash} 50

Appendix 4. Example test script

Below is a step by step walkthrough of one of the POC's test scripts.

Title: *Activate Panic Button – MGP Gameview*

Description: This test case navigates to a MGP game view and activates the panic button. In the beginning of the test case, a list containing poker and casino self-exclusion end dates is generated using the backend API calls. A similar list is generated in the end of the test case. It is also verified that the pop-up appearing when activating the self-exclusion is correct. The test case fails if:

- The two lists generated are alike i.e. activating the panic button has had no effect on the self-exclusion periods
- The content of the self-exclusion pop-up does not match the specification 100%

The test case can naturally fail due to multiple other reasons (missing elements at any point of the test) but the two conditions described are the focus of the test case.

1. **Keyword/Function:** *Set Up Device* **Arguments:** *Nexus 7*

Description: This Python call is a mandatory element prior to any test case. Within this keyword all SeeTest Automation's configurations are generated. The function takes as a mandatory parameter the device name on which the test case is run. In this example, the test is executed on Google's Nexus 7. By default, the function opens a new session in the casino client. It can be specified that the casino client will not be opened by using the optional argument open_casino_stg=='False' if for some reason it is not required. Within the proof of concept phase the second possible device is an iPad3.

2. **Keyword/Function:** *Verify client* **Arguments:**

Description: This Python call verifies that the casino client has opened correctly. The call passes if both *My menu* button (search based on its id) and text "ray.fi pelit" is found on the front page. The absence of the text snippet usually indicates that a player is already logged into the service. Therefore, if the text snippet is not found, a *Log out* Python call is being made. It performs a log out in the lobby view. Latest at this step the original step fails if the My menu button is still missing.

3. **Keyword/Function:** *IPJ – Generate self-exclusion data*
   Arguments: *(Information of a excludable player): Username, Passwordhash,*
   Return values: *A list containing two items: Casino self-exclusion end date and Poker self-exclusion end date.*

Description: This keyword resides in a pre-existed library. It makes an API call to the backend systems for player's self-exclusion information. The list is saved in a variable. The keyword fails only if the connection to the backend is broken or if the data retrieved is corrupted.

4. **Keyword/Function:** *Log in Mobile* **Arguments:** *Username, Password, Player's full name*

Description: This is a user keyword created in RF. It is created to simplify RF scripts as the functionality is used in most automated test cases. It calls for three different Python functions. Firstly, it calls for *Click My Menu Lobby* function. This function verifies that *My menu* button is visible in the screen, clicks on it and verifies that the username input

box has appeared. Secondly, a Python function *Insert Credentials* is called upon. It inserts player's username and password to the respective input boxes and clicks the log in button. Last Python call made is *Verify Text Found*. It is a generic call that verifies that the string given as an argument is found on the screen. In this case, player's full name is searched to verify that the login has been successful (player's full name is visible in the lobby view if logged in). The whole keyword fails if any element described earlier is not found.

5. **Keyword/Function:** *Open game* **Arguments:** *Kulta-Jaska*

Description: This Python call swipes the screen as long as the desired game is found on the screen and clicks on it when found. It uses game IDs to open the games. The game name is nevertheless given in natural language. Therefore, a game name to game ID library is implemented in the function. In this case, a popular game Kulta-Jaska is used. By default, the game is opened in real mode. By using the optional argument mode=='FUN' the game can be opened in fun mode. Lastly, the devices orientation is changed to match the game's requirements as some games are played in portrait mode and some in landscape. To match this requirement, all games are listed in which mode they belong to. Based on these two lists, the correct call is made. This function fails if any of the elements is not found.

6. **Keyword/Function:** *Verify game* **Arguments:**

Description: This Python call verifies that the game has opened. Firstly, a canvas element is searched for. The canvas element is present in most games on Android devices. It indicates that the player should swipe up the screen to activate full screen mode (get rid of Chrome's top banner). If the element is found, the screen is swiped up. All games open with these actions except for Roulette where a click to play element appears at this point. For one second, the element is searched for and clicked if found. If it is not found, nothing happens. Otherwise the next element to be found is the player's balance element in the game view based on its ID. It is an adequate measure to verify that the game opened correctly.

7. **Keyword/Function:** *Navigate Mgp* **Arguments:** *panicbutton*

Description: This Python call is used to navigate in the MGP game view menu. As there are two views of the MGP menu that differ a little, the function is somewhat complex. In the more common case, the menu appears by swiping from the upper left corner towards right. In the second case, a few action buttons are visible in the game view itself on top of the menu button. If the button that is to be clicked is not already on the screen, the menu button is clicked. In either case, the action button that is to be clicked is identified based on its ID and clicked. If the button is not visible on the screen, the menu view is swiped to locate the button. In this case, the panic button is clicked in the menu. The function fails if any of the elements to be clicked is absent.

8. **Keyword/Function:** *Verify Panicbutton Popup* **Arguments:**

Description: This Python call verifies that the pop-up appearing when clicking the panic button has correct content. In practice, it searches for a sting object matching the specifications on the screen. The function fails if the content of the pop-up is not exactly correct.

9. **Keyword/Function:** *Approve Panicbutton* **Arguments:**

Description: This Python call approves the panic button by clicking on the approval button identified based on its content. It also verifies that the approval pop-up appearing

has correct content and closes it from the *close* button identified based on its ID. The function fails if any of the elements is not found or the content of the approval pop-up is not correct.

10. Keyword/Function: *Navigate MGP* **Arguments:** *logout*

Description: The same Python call as in step 7 but this time the *log out* button is being clicked.

11. Keyword/Function: *Verify Text Found* **Arguments:** *ray.fi pelit*

Description: This is the same call already made within step 4. String "ray.fi pelit" is searched for. The string is not present if player is logged in in the lobby view. Thus, if the string is present, it is a good enough verification that the logout was successful and the player was navigated to the lobby view. The function fails only if the sting object is not found on the screen.

12. Keyword/Function: *IPJ – Generate self-exclusion data* **Arguments:** *As in step 3*
Return values: *As in step 3*

Description: A similar list as in step 3 is generated. It is saved in a separate variable.

13. Keyword/Function: *Run keyword FAIL if*
Arguments: *Lists generated in steps 3 and 12 match*

Description: This keyword is inbuilt in RF. It fails if the condition set in the argument is true. In this case, the lists generated in steps 3 and 12 are compared. If the lists match (the end date of the self-exclusion did not change within the test's execution) it is a clear sign that approving the panic button either failed or it did not have any effect on the end date of the self-exclusion.